VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA STROJNÍHO INŽENÝRSTVÍ
ÚSTAV AUTOMATIZACE A INFORMATIKY

FACULTY OF MECHANICAL ENGINEERING
INSTITUTE OF AUTOMATION AND COMPUTER SCIENCE

REALIZACE VYBRANÝCH VÝPOČTŮ POMOCÍ GRAFICKÝCH KARET

THE REALIZATION OF SELECTED MATHEMATICAL COMPUTATIONS USING GRAPHICAL CARDS.

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. PETR SCHREIBER
AUTHOR

VEDOUCÍ PRÁCE — Ing. VÍT ONDROUŠEK
SUPERVISOR

BRNO 2010

Vysoké učení technické v Brně, Fakulta strojního inženýrství

Ústav automatizace a informatiky
Akademický rok: 2009/2010

ZADÁNÍ DIPLOMOVÉ PRÁCE

student(ka): Bc. Petr Schreiber

který/která studuje v **magisterském navazujícím studijním programu**

obor: **Aplikovaná informatika a řízení (3902T001)**

Ředitel ústavu Vám v souladu se zákonem č.111/1998 o vysokých školách a se Studijním a zkušebním řádem VUT v Brně určuje následující téma diplomové práce:

Realizace vybraných výpočtů pomocí grafických karet

v anglickém jazyce:

The realization of selected mathematical computations using graphical cards.

Stručná charakteristika problematiky úkolu:

Paralelní výpočty se v současné době dostávají stále více do popředí jako účinný způsob získání vyššího výpočetního výkonu. Vhodný prostředek pro realizaci paralelních výpočtů představují grafické karty, jejichž čipy umožňují mnohem větší paralelismus ve zpracování úloh než CPU. Ze strany výrobců grafických karet se dočkáváme stále lepší podpory pro tyto účely, ať už se jedná o technologii nvidia Cuda, ATI stream či univerzálnější OpenCL. Tato práce je zaměřena na praktickou realizaci vybraných paralelních výpočtů pomocí GPGPU a jazyka OpenCL.

Cíle diplomové práce:

1, Proveďte rešeršní studii současného stavu problematiky výpočtů pomocí GPGPU,
2, Popište klíčové vlastnosti jazyka OpenCL a způsob jeho využití pro výpočty na grafických kartách,
3, Realizujte dynamickou knihovnu umožňující provádět výpočty vybraných matematických problémů na grafických kartách,
4, Porovnejte rychlost výpočtu Vámi realizovaného řešení pomocí GPGPU a řešení realizovaného na CPU.

Seznam odborné literatury:

Aaftab Munshi, The OpenCL Specification, 2009
OpenCL programming Guide for the CUDA architecture, Nvidia Co., 2009
Nvidia OpenCL JumpStart Guide, Nvidia Co., 2009

Vedoucí diplomové práce: Ing. Vít Ondroušek

Termín odevzdání diplomové práce je stanoven časovým plánem akademického roku 2009/2010.

V Brně, dne 12.11.2009

L.S.

Ing. Jan Roupec, Ph.D.	prof. RNDr. Miroslav Doupovec, CSc.
Ředitel ústavu	Děkan fakulty

ABSTRAKT

Práce pojednává o současných možnostech využití grafického hardware jakožto platformy pro provádění paralelních výpočtů. Text se zaměřuje na novou technologii OpenCL, která umožňuje pomocí stejného vysokoúrovňového kódu plně využít potenciálu vícejádrových procesorů a moderních grafických karet, bez vazby na specifického výrobce či operační systém. Autor předkládá čtenáři knihovny a nástroje založené na OpenCL spolu s praktickými příklady a vlastními postřehy ohledně současného stavu této technologie.

ABSTRACT

This work discusses available approaches for programming graphic hardware as a platform for executing parallel calculations. Text of the work is focused on new OpenCL technology, which allows executing the same high level code for taking control of full potential of multicore CPUs and GPUs, without explicit bindings to hardware vendor or operating system. Author provides the reader with libraries and tools based on OpenCL, along with practical examples and own observations about the current state of mentioned technology.

KLÍČOVÁ SLOVA

grafická karta, paralelní výpočet, GPGPU

KEYWORDS

graphic card, parallel computation, GPGPU

PODĚKOVÁNÍ

Chtěl bych touto cestou poděkovat všem, kteří mi radou, pravidelnými konzultacemi, nebo zapůjčením či zprostředkováním hardware umožnili práci dokončit.

Mé díky tak mají Ing. Vít Ondroušek, Ing. Pavel Houška, PhD, Jan Coufal, Jakub Vodrážka, Jakub Drábek, Petr Liška, Jiří Michalčík, Charles Pegge, Kent Sarikaya a mnozí další, na které jsem jistě, ne však úmyslně, zapomněl.

Velké díky pak patří i rodině, která mi byla po celou dobu studia značnou oporou.

OBSAH

1 ÚVOD

Navzdory překotnému vývoji v oblasti informačních technologií a příslušného hardware, i v 21. století se ve vědecko-technické praxi stále potýkáme s problémy, pro které výpočet řešení, díky své složitosti, trvá i na nejmodernějších procesorech často neúnosně dlouho.

Již od počátků programování počítačů člověk hledal cesty, jak dobu výpočtu zkrátit. Častým přístupem je intenzivní využívání práce s ukazateli či tvorba programu nebo jeho části v assembleru. S nástupem moderních procesorů pak přišly nové, moderní instrukce, umožňující v jednom kroku zpracovat souběžně více dat. Technologie jako MMX či SSE skutečně často přinesly a přinášejí znatelné zrychlení programu, díky datové mikroparalelizaci.

S nástupem vícejádrových procesorů začal být zřejmý význam možnosti paralelizace výpočtu ve větším měřítku, přesto bylo jistým problémem všechna jádra procesoru vytížit tak, aby se skutečně plně využilo jejich potenciálu. I tak lze však říci, že vícejádrové procesory představují již dnes dobrou platformu pro tzv. úlohový paralelizmus. Otázka platformy pro datový paralelizmus se však zdála dlouhou dobu neřešitelnou, snad s výjimkou poněkud speciálního případu SIMD instrukcí, které ovšem z jistého pohledu představují velice triviální případ datového paralelismu.

V době nástupu prvních 3D akcelerátorů si málokdo dokázal představit, že by se tato úzce specializovaná technologie jednou stala platformou pro obecné výpočty. Přesto se tak stalo, a grafické karty, dnes označované jako GPGPU, jsou schopny provádění data paralelních úloh. Technologii GPGPU lze stále považovat za relativně mladou, v mnoha ohledech snad i primitivní, ovšem na rozdíl od CPU, jehož vývoj se poslední roky zpomaluje, představuje GPU velký potenciál do budoucna.

Další rychlý vývoj v oblasti GPU výpočtů lze předpokládat už proto, že výpočtu schopné grafické karty se objevují i v obyčejných PC sestavách a noteboocích, a poptávka po využití jejich plného výkonu v oblasti práce s obrazem, urychlení kódování videa a dalších aplikacích stále roste i mezi obyčejnými uživateli PC. Dávno se již nejedná o technologickou raritu využívanou pouze ve vědeckých kruzích.

Předměty, zabývající se touto problematikou, jsou vyučovány na mnoha univerzitách ve světě. Je proto důležité nezůstat pozadu.

Tato práce představuje současné nejpoužívanější technologie, které programování grafických karet umožňují, jako jsou AMD Stream, Nvidia CUDA či DirectCompute. Těžištěm je však průzkum nové technologie OpenCL, která se od ostatních přístupů v některých podstatných ohledech liší.

Teoretická část nejdříve obecně rozebírá architekturu OpenCL, cíle, které si klade i specifika jazyka OpenCL C. Cizojazyčných publikací o programování GPGPU již v zahraničí několik vyšlo, v češtině se však programování grafických karet obecně či přímo pomocí OpenCL příliš zdrojů nevěnuje, proto si teoretická část klade za cíl představit technologii v pochopitelné formě, aby bylo možno udělat si představu o základních principech a způsobu využití.

Praktická část pak demonstruje několik řešených problémů z oblasti zpracování obrazu, jako jsou konvoluční filtry a segmentace obrazu. Dále se pak práce dotýká oblasti analýzy signálu a také mobilní robotiky, kde je použití GPGPU technologie vyloženě novinkou.

Zvláštní důraz je kladen na uvádění praktických postřehů a upozorňování na aktuální problémy současných implementací technologie i hardware na kterém běží, spolu s návrhy, jak nalezená úskalí řešit.

2 CÍLE PRÁCE

Zadání práce ukládá několik cílů, které je nutno splnit:

- Provedení rešeršní studie současného stavu problematiky výpočtů pomocí GPGPU
- Popsání klíčových vlastností OpenCL a způsob jeho využití pro výpočty na grafických kartách
- Realizace dynamické knihovny, umožňující provádět výpočty vybraných matematických problémů na grafických kartách
- Porovnání rychlosti výpočtu realizovaného řešení pomocí GPGPU a řešení realizovaného na CPU

Jelikož je práce úzce zaměřena na technologii OpenCL, bylo třeba prozkoumat možnosti její nejbližší konkurence. Za tu autor práce považuje v současnosti nejaktivněji vyvíjená řešení, kterým je věnován prostor v rámci kapitoly *Současný stav API pro programování GPGPU*. Pro lepší pochopení problematiky je však v kapitole *Raný vývoj programování grafických karet* nastíněn i sled technických inovací, které k možnostem provádění výpočtů, tak jak je známe dnes, vedly.

Druhý bod zadání, tedy prozkoumání OpenCL, jakožto perspektivní technologie nezávislé na hardware a operačním systému, je svým rozsahem větší, a snaží se o širší pohled na celou problematiku – od obecné architektury technologie, jazyka, který k programování zařízení používá až po některá její specifika, která musí mít programátor na paměti při vývoji aplikací.

Na základě poznatků, získaných v předešlých částech textu, je pak realizováno několik praktických řešení, založených na zmíněné technologii. Jedná se jak o realizovanou knihovnu pro práci s obrazem, tak i některé specializované nástroje, případně konkrétní samostatné výpočty, jejichž převod na GPU vedl k urychlení operace. Již v rámci této oblasti jsou uváděna porovnání výkonu mezi klasickou implementací problémů na procesoru a jejich analogickými verzemi na grafické kartě.

3 SEZNAM POUŽITÝCH POJMŮ

Tato kapitola slouží jako abecední seznam používaných odborných výrazů, technologií a zkratek. Většina názvů, i v dalším textu, je uvedena pod originálním anglickým názvem, jelikož zmíněné termíny, díky bouřlivému vývoji v oblasti GPGPU, nemají v češtině ekvivalent, nebo český výraz ještě není dostatečně zaveden. Některé z pojmů budou definovány průběžně v textu, ale pro snadnější přístup k popisu je možné najít je i v této kapitole.

Assembler. Nesprávné, ovšem často užívané označení jazyka symbolických adres. Assembler, správně *assembly language*, je nízkoúrovňovým, nestrukturovaným jazykem pro programování počítačů a obecně řešení založených na integrovaných obvodech.

AGP. Z angl. *Accelerated Graphics Port*. Starší typ sběrnice grafických karet, s teoretickou datovou propustností až 2133 MB/s. Výrobci tuto technologii v omezené míře stále podporují, v dnešní době je však již nahrazována výkonnější sběrnicí PCI-E.

Brooks, BrookGPU. Technologie vyvinutá na Standfordské univerzitě, umožňující programování i takových grafických karet, které nejsou označovány jako GPGPU. Do příchodu OpenCL preferovaný způsob programování karet fy ATi/AMD.

Cache. Typ vyrovnávací paměti, kompenzující rozdíly v rychlosti mezi dvěma výpočetními subsystémy. Donedávna typický prvek CPU, který se v poslední době začíná objevovat i na GPGPU.

CAL. Zkratka z angl. *Compute Abstraction Layer*. Nadstavbová technologie nad grafickými kartami fy ATi/AMD, umožňující programování GPGPU.

Cell. Mikroprocesor vyvinutý ve spolupráci firem IBM a Sony, představující mezistupeň mezi klasickými CPU a Stream procesory. Kromě vstupně výstupních obvodů obsahuje jedno hlavní jádro a osm koprocesorů.

Constant memory. V kontextu OpenCL podoblast *global memory*, která zůstává neměnná během vykonávání kernelu.

CTM. Zkratka z angl. *Close To Metal*. Technologie pro nízkoúrovňové programování grafických karet fy ATi/AMD.

CUDA. Soubor softwarových a hardwarových technologií, které firma Nvidia využívá k programování svých GPU. Více v kapitole *Nvidia CUDA*.

Datový paralelizmus. Typ paralelismu, zaměřený na distribuci dat mezi několika výpočetními jednotkami. V případě OpenCL pak mezi processing elements.

Direct Compute. Technologie fy Microsoft určená k obecnému programování grafických karet na platformě DirectX 11. Více v kapitole *DirectCompute*.

Dynamic branching. Datově závislé větvení, či cykly, které nemohou být v době kompilace optimalizovány rozbalením kódu. Technika je podporována až na moderních GPU, stále však může viditelně zpomalovat chod programu.

Fixed pipeline. Způsob využití grafické karty bez explicitního použití stream processorů. Přístup využívaný zejména před příchodem GPU a v aplikacích s nízkými nároky na vizuální věrnost.

GeForce. Označení mainstreamové série grafických karet fy Nvidia. Karty schopné obecných výpočtů jsou série 8000, 9000, GTn200, GTn300 a GTn400.

Global memory. V kontextu OpenCL oblast paměti, do které má přístup jak hostitel, tak všechny procedury spuštěné na výpočetním zařízení.

GPU. Zkratka z angl. *Graphics Processing Unit*. Termín původně definován firmou Nvidia jako „grafický čip s integrovanou hardwarovou transformací, nasvětlením, správou trojúhelníků a renderovacím engine, schopný vykreslení minimálně deseti milionů polygonů za sekundu“. Později byl tento termín chápán spíše jako libovolný grafický čip umožňující shader programming. V kontextu této práce je vzhledem k zaměření někdy této zkratky užito ve smyslu GPGPU.

GPGPU. Zkratka z angl. *General Purpose Graphics Processing Unit*. Tento termín je používán k označení grafických čipů, či grafických karet, které umožňují provádět obecné, tedy ne nutně pouze grafické, výpočty. Někdy je tato zkratka přímo využívána pro označení oblasti výpočtů na grafických kartách.

Kernel. V kontextu OpenCL i některých jiných technologií se kernelem rozumí procedura, účastnící se paralelního výpočtu. Kernel je možno využít pro realizaci úlohového i datového paralelizmu.

Local memory. V kontextu OpenCL oblast paměti s rychlým přístupem, která je sdílená mezi prvky pracovní skupiny.

OpenCL. Technologie pro programování heterogenního hardware nezávisle na operačním systému. Více v kapitole Technologie OpenCL.

MMX. Zkratka z angl. *MultiMedia Extensions*. Jedná se o historického předchůdce SSE, ovšem s omezením pouze na operace s celými čísly. Práce MMX instrukcí je prováděna nad osmi 64bitovými registry.

PCI. Z angl. *Peripheral Component Interconnect*. Dnes již nepoužívaný typ obecné sběrnice, s teoretickou datovou propustností až 133 MB/s. Pro potřeby GPU později nahrazena AGP.

PCI-E, PCIe. Z angl. *Peripheral Component Interconnect Express*.V současné době poslední model sběrnice využívané pro integraci grafické karty do základní desky. Nejvýkonnější varianta má teoretickou datovou propustnost až 16 GB/s.

Pozdní vazba. V kontextu práce s dynamickou knihovnou pod Windows se jedná o mechanismus volání procedury, kdy v okamžiku kompilace kódu není znám její ukazatel. Ten je získán až dodatečně, za běhu programu, pomocí volání *GetProcAddress*.

Private memory. V kontextu OpenCL oblast paměti přístupná pouze proměnným uvnitř každého kernelu, která však nemůže být sdílena mezi kernely navzájem. Analogií proměnné umístěné v private memory je v klasickém programování proměnná lokální.

Processing element. V kontextu OpenCL nejmenší jednotka schopná samostatně spouštět kernel. V případě data paralelního programování jsou většinou využity desítky až stovky processing elements.

Radeon. Označení mainstreamové série grafických karet fy ATi/AMD. Karty schopné obecných výpočtů jsou série HD 4000 a HD 5000.

Rasterizace. Proces převodu scény reprezentované vektorovými primitivy na dvourozměrnou bitmapu.

SIMD instrukce. Z angl. *Single Instruction Multiple Data*. Tímto názvem jsou souhrnně označovány instrukce, schopné provést jednoduchou operaci současně nad několika čísly. Typickým příkladem je např. sečtení komponent vektoru.

Shader programming. Programování grafických karet, umožňující definovat grafickým primitivům vlastní způsob vykreslování fragmentů, modifikovat jejich vrcholy a na základě jejich existující definice vytvářet primitiva nová.

SSE. Zkratka z angl. *Streaming SIMD Extensions*. Jedná se o specializované instrukce, které umožňují mikroparalelní práci nad daty. Instrukce SSE mohou zároveň vykonat stejnou operaci nad dvěma až šestnácti hodnotami. První verze této instrukční sady se poprvé objevila v procesorech Pentium III. V současné době umožňují práci jak s celočíselnými datovými typy, tak i operace s plovoucí desetinnou čárkou v jednoduché i dvojité přesnosti.

Stream. V kontextu této práce soubor softwarových a hardwarových technologií, které firma ATi/AMD využívá k programování svých GPU. Více v kapitole *ATi Stream*.

Stream procesor. Součást grafické karty, umožňující programátorovi vykonávat na GPU vlastní výpočty. Typický počet stream procesorů je v současné době od šestnácti do řádu stovek. Jeden stream processor může obsahovat více processing elements. Firma ATi obvykle označuje tímto termínem přímo processing element. U fy Nvidia je stream processor nazýván jako CUDA core, a typicky obsahuje 8 processing elements.

Textura. Tímto termínem je v textu označován souvislý blok dat v paměti grafické karty, reprezentující 2D či 3D obrázek. Datovým typem barevné komponenty může být jak celé číslo, tak i číslo s plovoucí desetinnou čárkou v jednoduché přesnosti.

Teselace. Proces zahušťování již existující polygonové sítě.

Úlohový paralelizmus. Typ paralelismu, zaměřený na rozdělení dílčích samostatných úloh mezi několika procesory. V případě OpenCL pak mezi processing elements.

4 OD GRAFICKÉHO 3D AKCELERÁTORU K GPGPU

4.1 Raný vývoj programování grafických karet

Za první běžně dostupný grafický akcelerátor je obecně považována karta 3DfX *VooDoo*, která se objevila již roku 1996. Přestože ve své době znamenala malou revoluci, nikoho by nenapadlo, že by se z tohoto typu zařízení v budoucnu mohlo vyvinout něco jako grafická karta pro obecné výpočty. Karta samotná fungovala pouze jako 3D akcelerátor, pro standardní 2D zobrazení bylo nutné mít navíc plnohodnotnou 2D grafickou kartu. *VooDoo* bylo připojeno přes datovou sběrnici PCI, která svou propustností kolem 133 MB/s do značné míry omezovala výměnu dat, nehledě na nízké 50 MHz taktování čipu na kartě. Paměť karty pouhých 4 MB taktéž zcela vylučovala zpracování velkého objemu informací. Tento grafický akcelerátor podporoval velice omezenou sadu funkcí, která však postačovala pro urychlení rasterizace virtuální scény. Toto řešení je možno z dnešního pohledu chápat jako velice úzce specializovaný koprocesor bez možnosti využití pro obecné programování.

V následujících letech byly karty zrychlovány, osazovány většími pamětmi, ovšem z hlediska potenciálu pro obecné programování se jednalo o nepodstatné změny. Je důležité poznamenat i to, že poměrně náročné operace, jakými je maticové násobení a transformace vektorově reprezentované scény se v této době stále prováděly na procesoru a akcelerátor mu v tomto nijak neulehčoval[1].

Do roku 2000 ještě došlo k několika zajímavým inovacím. První, která přiblížila akcelerátory k možnostem výpočetního využití bylo vyvinutí rychlejší, čistě grafickým kartám určené sběrnice AGP s teoretickou propustností až 2133 MB/s. Dalším krokem bylo představení grafické karty nového typu, první s označením GPU. Nvidia GeForce 256 nesla první znaky rozšířené programovatelnosti v podobě *Registry combiners*[1]. Ty umožňovaly provádět jednoduché, programátorem definované operace nad objekty textur. Jednalo se spíše o jakousi parametrizaci procesu s možností vyhodnocovat jednoduché výrazy. Výsledkem však byla změna textury, nejednalo se o obecný výpočet v pravém slova smyslu.

Dalším výrazným krokem vpřed, do fáze, kdy již lze alespoň hovořit o nějaké formě programu, určeného ke spuštění na GPU, bylo představení technologie DirectX 8.0 s podporou pixel a vertex shaderů. První kartou s podporou tohoto rozhraní byla GeForce 3, která v době svého vydání způsobila možností programovat některé operace pomocí assembleru podobného jazyka smíšené reakce.

```
ps.1.2

tex t0            ; práce bude provedena s texturou v registru t0
texm3x3pad t1, t0 ; vypočte u z prvního řádku
texm3x3pad t2, t0 ; vypočte v z druhého řádku
texm3x3 t3, t0    ; vypočte w z třetího řádku
                  ; uloží u, v a w do t3
mov r0, t3
```

Obr. 1 Úryvek kódu pro maticové násobení[2]

Sice se stále jednalo o výpočty čistě grafického charakteru, tato cesta však umožnila kompletně redefinovat chování některých částí procesu zpracování obrazu. S příchodem nového hardware a inovovaných způsobů se zápis kódu přesunul ze série jednoduchých instrukcí na vyloženě vysokoúrovňovou syntaxi, vycházející z jazyka C.

[1] Snad s výjimkou čipů postavených na platformě Rendition

Tento způsob programování již je většinou akceptován jako dostatečně přehledný a začínají se objevovat první pokusy o využití shader technologie pro provádění výpočtů s obecným zaměřením.

Moderní vzhled zápisu programu pixel shaderu v HLSL na Obr. 2 ostře kontrastuje s původní podobou shaderů na Obr. 1.

```
sampler2D input : register(s0);
float4 main(float2 uv : TEXCOORD) : COLOR
{
    float4 Color;
    Color = tex2D( input , uv.xy);
    return Color;
}
```

Obr. 2 Šablona pixel shaderu[3]

Jedním z projektů, který se zaměřil na možnosti programování GPU již v roce 2004 byl *Brooks*, vyvinutý na Standfordské univerzitě[4]. Způsob, jakým obešel omezení tehdejší doby, a to grafice specifický zápis programu, je elegantní. Vysokoúrovňový kód se syntaxí podobnou C je, zjednodušeně řečeno, transformován tak, aby jeho výstupem byl vertex/pixel shader, který je možno dále zkompilovat pro GPU pomocí driveru. BrookGPU tak lze chápat jako abstraktní vrstvu nad klasickým programováním shaderů.

```
kernel void krnIntersectTriangle(Ray ray<>, Triangle tris[],
                                 RayState oldraystate<>,
                                 GridTrilist trilist[],
                                 out Hit candidatehit<>)
{
  float idx, det, inv_det;
  float3 edge1, edge2, pvec, tvec, qvec;

  if(oldraystate.state.y > 0)
  {
    idx = trilist[oldraystate.state.w].trinum;
    edge1 = tris[idx].v1 - tris[idx].v0;
    edge2 = tris[idx].v2 - tris[idx].v0;
    pvec = cross(ray.d, edge2);
    det = dot(edge1, pvec);
    inv_det = 1.0f/det;
    tvec = ray.o - tris[idx].v0;
    candidatehit.data.y = dot( tvec, pvec ) * inv_det;
    qvec = cross( tvec, edge1 );
    candidatehit.data.z = dot( ray.d, qvec ) * inv_det;
    candidatehit.data.x = dot( edge2, qvec ) * inv_det;
    candidatehit.data.w = idx;
  }
  else
  {
    candidatehit.data = float4(0,0,0,-1);
  }
}
```

Obr. 3 Ukázka kódu v BrookGPU[4]

Na Obr. 3 je uveden výpis kódu v BrookGPU. Je dobré si vzhled tohoto kódu zapamatovat pro pozdější srovnání s OpenCL C, které, jak uvidíme, sdílí některé prvky.

4.2 Současný stav API pro programování GPGPU

Na přelomu let 2006/2007 se začaly objevovat nové modely grafických karet, u kterých se jejich využití pro obecné výpočty dostávalo do popředí zájmu a začalo být i otevřeně propagováno výrobci hardware.

V následujícím textu se tak seznámíme s technologiemi, které se od té doby vyvíjely a jsou stále dostupné a používané. Jedinou významnou technologií, která v tomto stručném přehledu chybí, je OpenCL, které je však podrobně rozebráno ve zbytku práce

4.2.1 Nvidia CUDA

Ke konci roku 2006 byla uvedena ve své době převratná grafická karta GeForce 8800, která obsahovala několik vylepšení oproti předchozím architekturám. Jak bylo dříve uvedeno, grafické karty byly programovatelné pomocí shader programů, z nichž část se věnovala pouze oblasti ovlivnění výsledné barvy pixelu, zato druhý typ zpracovával vrcholy geometrie.

Dříve byla každá z těchto operací prováděna na úzce specializovaných jednotkách, a tak se často stávalo, že jednotky pro zpracování pixelů byly maximálně vytíženy, zatímco jednotky pro úpravu vrcholů byly z větší části nečinné. Tento fakt vedl u nové série 8000, do které spadá i uvedená karta, k odlišnému přístupu, který spočíval v obecnějším návrhu shader jednotek tak, aby mohly zpracovávat libovolnou úlohu. Tímto byla zároveň otevřena cesta ke zcela obecnému využití těchto pomocných procesorů, např. pro provádění libovolných výpočtů.

Firma Nvidia tak velice záhy začala s vývojem vlastní technologie, která by umožnila programování jejich GPGPU (General Purpose Graphics Processing Unit). Technologie byla nazvána CUDA a jedná se o zkratku z angl. *Compute Unified Device Architecture*. Tímto souhrnným označením rozumíme soubor hardwarových i softwarových prostředků určených pro provádění obecných výpočtů na grafických kartách.

Samotné programování je prováděno pomocí kombinace run time API, které používá programátor z libovolného programovacího jazyka pro kompilaci a spouštění programů na GPU. Jazyk, ve kterém jsou programy psány, může být dvojí. Tím prvním, nejčastěji prezentovaným způsobem, je zápis kódu v CUDA C. Tento jazyk vychází z jazyka C, oproti němu však poskytuje jistá rozšíření. Zápis programu je možno shlédnout na Obr. 4.

```
__global__ void MyKernel(float* devPtr, int pitch)
{
    for (int r = 0; r < height; ++r)
    {
        float* row = (float*)((char*)devPtr + r * pitch);
        for (int c = 0; c < width; ++c)
        {
            float element = row[c];
        }
    }
}
```

Obr. 4 Ukázka kódu v CUDA C[5]

CUDA C však není jediným jazykem, který je v případě karet fy Nvidia použít. Kromě DirectCompute a OpenCL, o kterých bude podána informace ve vyhrazených kapitolách, se jedná i o CUDA Fortran. Tento jazyk, který byl navržen již v 50. letech

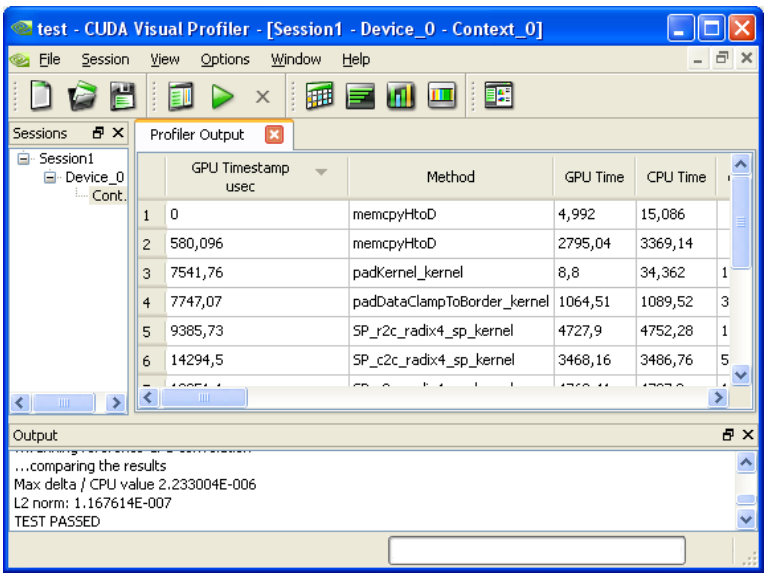
minulého století tak je stále používán, byť v pozměněné podobě, jak ukazuje zápis GPU programu na Obr. 5.

```
attributes(global) subroutine ksaxpy( n, a, x, y )
    real, dimension(*) :: x,y
    real, value :: a
    integer, value :: n, i

    i = (blockidx%x-1) * blockdim%x + threadidx%x
    if( i <= n ) y(i) = a * x(i) + y(i)
end subroutine
```

Obr. 5 Ukázka kódu v CUDA Fortran[6]

Technologie CUDA je v současné době dostupná ve verzi 3.0, která již dokonce umožňuje programovat GPU kód objektově, pomocí C++. Tato možnost se však týká až nejnovějších modelů grafických karet této firmy, které jsou však, jako novinka, zatím dostupné za poměrně dost vysokou cenu.



Obr. 6 Uživatelské rozhraní CUDA Visual Profiler

Součástí platformy CUDA je i sada pomocných nástrojů, které usnadňují vývoj aplikací, jakým je například dodávaný profiler pro analýzu výkonu. V současné době je ve vývoji i integrace ladění CUDA aplikací přímo z prostředí Microsoft Visual Studia.

Za nevýhodu této technologie lze považovat její úzkou vazbu na hardware firmy Nvidia. Programy na platformě CUDA tak nelze provozovat na grafických kartách jiných výrobců. V případě fy Nvidia jsou pro obecné výpočty podporovány karty série 8000, 9000, GTn200, GTn300 a GTn400.

Pro některé aplikace tento fakt však patrně nepředstavuje vážnější překážku. Již dnes je možné pozorovat aplikace CUDA v různých oblastech, od zábavního průmyslu[7][8] až po lékařství[9].

Jistou výhodou je pak dostupnost několika tištěných knih[10], které programátora uvádějí do problematiky, výjimkou však nejsou i popisy řešení specifických úloh. Zmíněné knihy jsou však v angličtině či ruštině, české překlady se patrně díky úzké specializaci u nás v brzké době neobjeví.

Některé z původně placených tištěných publikací, věnovaných programování karet fy Nvidia, jsou nyní dostupné online[11].

O úspěchu zavedení této technologie hovoří i fakt, že je vyučována na několika univerzitách v zahraničí, mimo jiné na *Harvard University, University of Illinois at Urbana-Champaign* nebo *Chinese Academy of Sciences*.

4.2.2 ATi Stream

Další neméně důležitou technologii představuje řešení ATi Stream, někdy označované jako AMD Stream či ATi/AMD Stream, stále se však jedná o stejnou technologii a proto bude v následujícím textu označována prvním uvedeným způsobem.

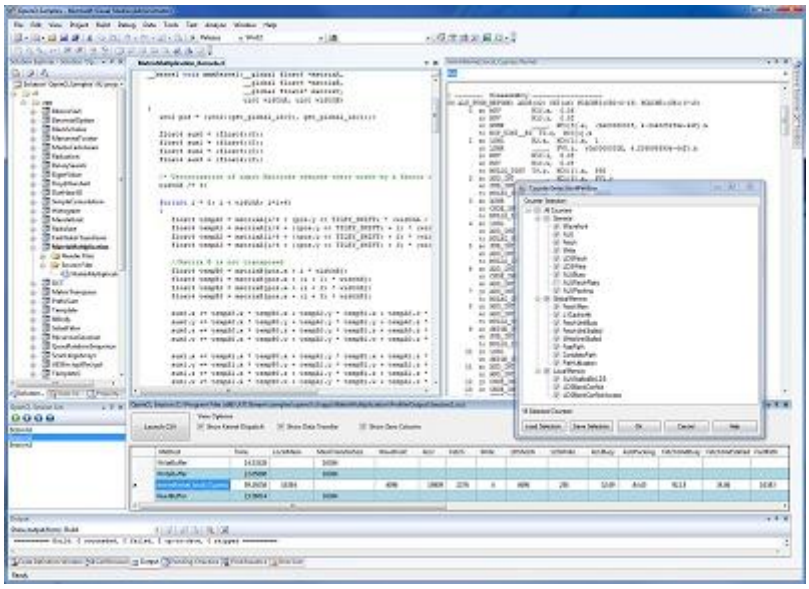
Jak již název napovídá, jedná se, podobně jako v případě Nvidia CUDA, o technologii úzce vázanou na hardware jedné firmy.

Zatímco v případě fy Nvidia bylo již od počátku programování prováděno přes jazyk podobný C, v případě ATi byl vývoj o něco bouřlivější, a tak název ATi Steam v různých obdobích označoval různé přístupy, všechny však vedly k programování grafických karet.

Bezprostředním předchůdcem této technologie je *Data Parallel Virtual Machine*, později označované jako CTM, které je někdy přímo považováno za raného představitele nástrojů ATi Stream. Přínosem CTM, z angl. *Close To Metal*, bylo zprostředkování přístupu k funkcionalitě stream procesorů bez nutnosti využívat k tomu grafické jazyky pro definici shaderů.

Nejednalo se však o vysokoúrovňový jazyk, ale podobně jako v případě prvních verzí shaderů o definici kódu velice blízkou assembleru. Přestože byla tato technologie pozvolna zaváděna i na některých vysokých školách např. na *Instituto Superior Técnico University* v Portugalsku, byl tento projekt záhy ukončen, a nahrazen novou technologií *Compute Abstraction Layer*.

Nad touto vrstvou byla později vystavěna verze již dříve představeného *BrooksGPU*, v souvislosti s ATi někdy označovaného jako *Brooks+*.



Obr. 7 Uřivatelské rozhraní ATi Stream Profiler[12]

ATi Stream však nejsou jen jazyky, kterými je GPU programováno, celá tato technologie poskytuje programátorovi i několik nástrojů pro usnadnění vývoje. Jak ukazuje Obr. 7, ATi Stream Profiler dává programátorovi možnost analýzy jeho kódu

v přehledně rozvrženém uživatelském prostředí, včetně možnosti sledování překladu vysokoúrovňového kódu na strojové instrukce.

V současné době je u ATi Stream patrný odklon od *Brooks+* k nové technologii OpenCL. Té je však věnována dostatečná pozornost v kapitole *Technologie OpenCL*, a proto se jí pro tuto chvíli nebudeme zabývat.

Pro lepší přehled je nutné dodat, že různými přístupy lze pomocí ATi Stream programovat všechny karty HD série, ovšem podpora OpenCL je omezena na poslední dvě modelové řady, tedy HD 4000 a HD 5000.

4.2.3 DirectCompute

Poslední technologie zmíněná v této kapitole pochází od fy Microsoft. Na rozdíl od Nvidia CUDA a ATi Stream již není úzce vázána na hardware konkrétní firmy, a omezení pro ni platí v podstatě dvojí. Tím prvním je provoz z operačního systému Windows, a to pouze ve verzích Vista a 7, druhým je pak podpora pouze na hardware schopném provozu DirectX 10 a 11. Toto omezení je však zcela v souladu s požadavky všech ostatních moderních výpočetních API.

DirectCompute bylo zavedeno v roce 2009 jako součást multimediálního balíku DirectX 11, a schopnost provádění výpočtů je v rámci této technologie označována jako *compute shader*. Zajímavostí je pak rozdíl ve výkonnosti při použití DirectX 11 hardware, kdy dochází často až k trojnásobnému vylepšení výkonu[13] oproti hardware pro DirectX 10, díky sdílené paměti mezi výpočetními jednotkami. Tuto informaci doporučuje brát autor práce s rezervou, protože např. karty fy Nvidia mají tento typ paměti implementován i na starším hardware.

Jak již bylo částečně naznačeno, velkou výhodou je použitelnost této technologie na grafických akcelerátorech obou předních firem, tedy ATi a Nvidia. Na první platformě je DirectCompute implementováno pomocí vrstvy CAL, na platformě Nvidia pak jako nádstavba nad CUDA.

```
struct BufferStruct
{
  uint4 color;
};

cbuffer consts {
  uint4 const_color;
};

cbuffer veryconsts {
  uint4 very_const_color;
};

// group size
#define thread_group_size_x 4
#define thread_group_size_y 4
RWStructuredBuffer<BufferStruct> g_OutBuff;

[numthreads( thread_group_size_x, thread_group_size_y, 1 )]

void main( uint3 threadIDInGroup : SV_GroupThreadID,
           uint3 groupID : SV_GroupID,
           uint groupIndex : SV_GroupIndex,
           uint3 dispatchThreadID : SV_DispatchThreadID )
{
        int N_THREAD_GROUPS_X = 16;
        int stride = thread_group_size_x * N_THREAD_GROUPS_X;
```

```
        int idx = dispatchThreadID.y * stride + dispatchThreadID.x;

        uint4 color = uint4( groupID.x, groupID.y , const_color.x,
very_const_color.x );

        g_OutBuff[ idx ].color = color;
}
```

Obr. 8 Ukázka kódu compute shaderu[14]

Podobně, jako ostatní API, umožňuje i tato technologie provádění paralelních výpočtů, které lze organizovat do menších skupin, ve kterých je pak garantováno souběžné vykonávání vláken.
Stejně jako všechny moderní shadery, jejichž jazyk je na platformě DirectX označován jako HLSL, tedy *High Level Shading Language*, je program zapsán ve srozumitelné, jazyku C podobné formě, jak ukazuje Obr. 8.

5 TECHNOLOGIE OPENCL

Rozhraní OpenCL bylo původně navrženo firmou Apple v roce 2008. První ucelený návrh byl předán organizaci Khronos, která má ve svém portfoliu i další standardy, jako je např. OpenGL. Khronos ve spolupráci s dalšími výrobci hardware a software došel ke konečné specifikaci OpenCL 1.0, dostupné o rok později[15].

OpenCL je zaměřeno na paralelní programování, podobně jako řešení zmíněná v minulé kapitole. Na rozdíl od zmíněných API však přineslo několik nových myšlenek - nezávislost na operačním systému, absence úzké vazby na grafický hardware a s tím i možnost provádět paralelní výpočty na moderních vícejádrových CPU.

OpenCL tvoří tří základní části:

- Správa platforem
- Run time API
- Jazyk OpenCL C

V následujících kapitolách bude vysvětleno k čemu ta která část slouží a jak přispívá k možnosti provádět paralelní výpočty. Celá práce se věnuje OpenCL ve verzi 1.0, některá omezení a znaky platné pro tuto verzi tak nemusí platit pro novější revize technologie, které však v době vyhotovení práce ještě nebyly dostupné.

5.1 Správa platforem

Tato vrstva umožňuje zjistit, zda jsou na dané konfiguraci dostupné platformy schopné provádět výpočty pomocí OpenCL. V současné době se platformy dělí na GPU, CPU a tzv. *OpenCL akcelerátor*.

Platformou typu GPU je moderní grafická karta s tzv. stream procesory. V současné době se jedná o modely běžně dostupných sérií Nvidia GeForce 8 a novější, ATi Radeon HD 4000 a novější a poměrně překvapivě i karty od fy S3, které jsou u nás ovšem obtížně sehnatelné.

Platformou typu CPU je teoreticky libovolné vícejádrové CPU s podporou alespoň SSE3 instrukcí, třída *OpenCL accelerátor* je potom reprezentována specializovanými kartami do PCI-E.

Základní myšlenkou OpenCL je možnost využití jednoho i několika zařízení, i různých typů, zároveň, dle uvážení programátora aplikace.

Správa platforem slouží i k vytvoření tzv. kontextu zařízení, který určuje, které zařízení bude využito v programu, a na tento kontext jsou pak vázány konkrétní operace se zařízením. Na tomto místě by bylo vhodné vysvětlit, jak je výpočetní zařízení chápáno v terminologii OpenCL.

Model OpenCL pracuje s pojmem *hostitele* a *OpenCL zařízení* (dále jen zařízení). Hostitel je prvkem, který určuje, které úlohy se budou na zařízeních spouštět. Hostitelem se tak stává libovolný program, který využívá OpenCL run time API. OpenCL zařízení, ke kterému si program kontext vytvoří, se pak stává platformou pro provádění výpočtů. OpenCL na toto zařízení nahlíží jako na soubor *computing units*, které jsou pak dále děleny na tzv. *processing elements*. Processing elements si lze představit jako jádra, na kterých může běžet jedna ucelená část výpočtu. V případě procesoru se jedná o jednotky, u GPU o desítky až stovky samostatných *processing elements*.

Pomocí zprávy platforem je možné získat i detailní informace o jednotlivých zařízeních jako je počet computing units, takt, obsluhovaná paměť a mnoho dalšího.

5.2 Run time API

Run time API je komponentou OpenCL, která umožňuje hostiteli komunikovat s jedním či více zařízeními. Toto rozhraní obstarává následující základní oblasti:

- Definice datových struktur pro výměnu dat mezi hostitelem a zařízením
- Práce s OpenCL programy
- Práci s kernely
- Správa fronty úloh
- Sledování výkonu

Jak napovídá první položka seznamu, paměť hostitele a zařízení není přímo automaticky synchronizována. Důvodů pro toto chování je více, tím nejzřejmějším je fakt, že automatická synchronizace paměti hostitele a zařízení by vedla k neustálému provozu na sběrnici, což by i v případě teoreticky velice rychlého PCI-E vedlo k výraznému zpomalení chodu programu. Proto je na programátorovi hostitelské aplikace, kdy pošle data na zařízení a kdy je přečte zpět.

K této výměně dat slouží v OpenCL speciální paměťové objekty, které dělíme na *buffery* a *image objects*. Bufferem je v souvislosti s OpenCL myšlena oblast dat, kterou si lze představit jako jednorozměrné pole. Image object je pak oblast v paměti, na kterou je možno nahlížet jako na 2D či 3D pole. Tento typ objektů je velice výhodně zpracovatelný na GPU, které je díky dlouhodobému zaměření na práci s texturami na tuto práci dobře připraveny. Díky tomuto faktu tak nabízejí např. možnost velice rychlé lineární interpolace mezi hodnotami, případně rychlý přístup na indexovaný prvek nejen 2D ale i 3D image bufferu. Tyto datové objekty je taktéž možno přímo sdílet s grafickou knihovnou OpenGL jako textury, a bez jakéhokoli přesunu zpět k hostiteli je rovnou vykreslovat na obrazovku či do frame bufferu.

Další oblastí, kterou se run time API zabývá, jsou tzv. *OpenCL programy* (dále jen programy). Tento název je ovšem trochu zavádějící, protože program se v OpenCL více podobá konceptu dynamicky linkované knihovny, jak ji známe ze systému Windows. Program je tedy kolekcí několika, třeba i zcela nezávislých procedur, které jsou sdruženy v jednom balíčku. OpenCL program může obsahovat dva typy procedur – tzv. *kernely* a pomocné funkce. Kernely rozumíme funkce, o jejichž vykonání může hostitel požádat pomocí run time API, zatímco procedury pomocné nejsou zvnějšku viditelné a jsou interně využívány pouze kernely. Více o nich v samostatné části o OpenCL C.

Spouštění úloh ke zpracování lze provádět jak synchronně, tak asynchronně a jednotlivé úlohy lze řadit do fronty, která je dále automaticky zpracovávána. Programátor se v případě asynchronních úloh může dotazovat na jejich aktuální stav a podle toho řídit výpočet.

Čas provádění operací na zařízení pak lze sledovat pomocí profilovacích funkcí, a to teoreticky až s přesností na nanosekundy.

5.3 Programování GPGPU

Dříve než bude popsán samotný jazyk OpenCL C, je třeba vysvětlit některé základní prvky obecného přístupu k programování GPU, z pohledu OpenCL jako takového. První věcí, kterou je důležité si uvědomit, je fakt, že GPU je vhodné zejména ke zpracování paralelních úloh, díky velkému počtu samostatných jednotek, na kterých mohou souběžně výpočty běžet.

Zatímco na CPU lze již delší dobu sledovat implementace *úlohového paralelizmu,* ke kterému postačuje relativně malý počet jader, největším přínosem GPU je oblast paralelizmu datového. Na GPU, díky velkému počtu *processing elements,* je

možno stejnou práci nad konkrétními daty velice jemně rozložit na samostatné jednotky. Processing elements na GPU jsou v mnoha ohledech primitivnější, než jádra CPU, např. co se týče velikosti paměti, kterou mohou obsloužit, ale jejich množství tento nedostatek v případě použití na datový paralelizmus nahrazuje.

Úlohový paralelizmus je jednoduché si představit jako rozdělení úlohy na jednotlivé samostatné celky. Pro ilustraci - v případě robotu sběr dat ze senzorů, lokalizace, plánování. Paralelizmus datový není tak obvyklý, a proto si dovoluji uvést několik příkladů pro objasnění jeho principu a použití na několika vzorových úlohách.

V případě součtu dvou velkých polí vektorů standardními postupy se nelze vyhnout nějaké formě cyklu.

```
for (i=0; i<pocet; i++)
{
  c[i].x = a[i].x + b[i].x;
  c[i].y = a[i].x + b[i].y;
  c[i].z = a[i].z + b[i].z;
  c[i].w = a[i].w + b[i].w;
}
```

Obr. 9 Typický kód pro součet komponent vektoru

Z uvedeného modelového případu na Obr. 9 lze vypozorovat, že položka s indexem *i+1* bude vždy čekat na vyhodnocení, dokud nejsou vypočítány všechny předchozí, přestože na nich nijak nezávisí. Taktéž je vidět, že v jednom pracovním cyklu tak vyhodnotíme pouze jeden celý součet dvou vektorů.

Na GPU by bylo jednoduše možné spouštět program, který by neobsahoval cyklus žádný a odpovídal by kódu uvnitř cyklu, jak ukazuje Obr. 10.

```
  i = get_global_id[0];

  c[i].x = a[i].x + b[i].x;
  c[i].y = a[i].x + b[i].y;
  c[i].z = a[i].z + b[i].z;
  c[i].w = a[i].w + b[i].w;
```

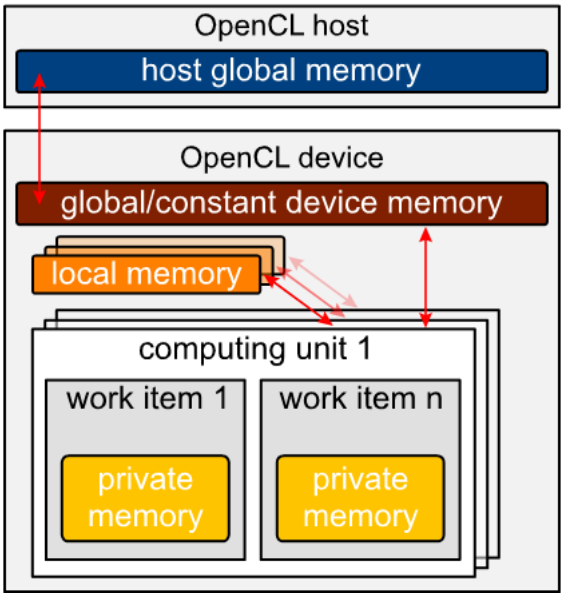
Obr. 10 Jedna z možných variant kódu pro součet komponent vektoru v OpenCL

Program samotný si pouze před součtem zjistí, který index pole obsluhuje. Na grafické kartě s např. 240 processing elements by tak v jednom pracovním cyklu bylo zároveň sečteno 240 položek pole, oproti pouhé jedné položce v klasickém programu na CPU. Velkou výhodou OpenCL je fakt, že samo obstará vytížení všech processing elements, dokud není výpočet ukončen. Pokud tedy sčítáme pole o 24 000 položkách na GPU s 240 processing elements, není nutné výpočet v režii programátora dělit na 100 samostatných volání. Práce vykonávaná jedním processing element je označována jako *work-item*.

Nutno dodat, že tento příklad představuje ideální stav. Většinou na sebe totiž elementy výpočtu (v tomto případě položky pole) nějakou vazbu mají. V takovém případě lze problém rozdělit na *pracovní skupiny* (work groups), které mohou efektivně komunikovat a sdílet data.

Na tento problém se úzce váže model paměti v OpenCL. Pomineme-li paměť hostitele, který úlohy na zařízení spouští, OpenCL definuje 4 základní typy paměti, ke kterým má program na GPU přístup. Pro lepší pochopení je model paměti ilustrován na Obr. 11. Terminologie této části OpenCL může být pro programátora zvyklého na

termíny používané ve vysokoúrovňovém programování na PC poněkud zavádějící a to z toho důvodu, že známé názvy označují v OpenCL zcela jiné fenomény, což platí dokonce i v porovnání s CUDA C, které je jazykem taktéž určeným pro GPU. I z tohoto důvodu je text na Obr. 11 uveden v anglickém jazyce.



Obr. 11 Schéma pojetí paměti v OpenCL[15]

První typ paměti se nazývá *soukromá paměť* (private memory) a je to paměť, se kterou se pracuje v rámci procedury. *Lokální paměť* (local memory) pak v OpenCL označuje paměť, kterou sdílí procedury v rámci jedné pracovní skupiny. Dalším typem oblasti paměti je *neměnná paměť* (constant memory), která označuje neměnný blok paměti, určený pouze ke čtení. Posledním typem paměti je *paměť globální* (global memory), která ovšem není pamětí sdílenou mezi hostitelem a OpenCL zařízeními, ale pouze pamětí globálně přístupnou v rámci jednoho OpenCL zařízení.

V současné době není možné sdílet paměť mezi více zařízeními, případnou kooperaci mezi zařízeními pak zajišťuje hostitel, který může data ve vlastní režii kopírovat a zapisovat na daná zařízení.

Tolik tedy velmi stručně k základním principům, které je třeba mít na paměti při programování GPU. Nyní se podíváme blíže na jazyk, který k programování grafických karet využíváme.

5.4 OpenCL C

Programování úloh, které běží na jednotlivých zařízeních, je prováděno nezávisle na typu zařízení vždy pomocí OpenCL C. Jak již název napovídá, tento programovací jazyk vychází z jazyka C, konkrétně ze standardu C99. Nejedná se však o jeho exaktní kopii, jelikož některé vlastnosti vynechává, a naopak přidává vlastnosti nové, zejména v oblastech dotýkajících se paralelizmu.

5.4.1 Datové typy a proměnné

Co se týče skalárních datových typů, je podporována standardní sada celočíselných (8, 16, 32 a 64 bit) i typů s plovoucí desetinnou čárkou. V případě typů s plovoucí desetinnou čárkou je zaveden i datový typ *half*, který je pouze 16 bitový. Tento zjednodušený datový typ má uplatnění spíše v aplikacích zábavního průmyslu, kde snížená přesnost nevadí, a je třeba šetřit pamětí (např. animování modelu).

Datové typy 64bitové jsou ve specifikaci pokryty, ale nejsou důsledně vyžadovány. Tento krok je patrně důsledkem současného stavu nabídky grafického hardware, kde je dvojnásobná přesnost podporována jen u posledních, či drahých modelů a výkon je většinou oproti výkonu v jednoduché přesnosti značně degradován. GPGPU hardware schopný rychlého zpracování operací v dvojnásobné přesnosti se pozvolna začal objevovat až v průběhu roku 2010, v podobě grafických karet Nvidia založených na Fermi architektuře [16].

Novinkou v OpenCL C jsou pak zabudované vektorové datové typy, které vznikly složením z dostupných typů skalárních. Vektory v OpenCL C mají 2, 4, 8 nebo 16 komponent, pro optimální zarovnání v paměti. Na vektorové typy se pak váže i změna v syntaxi a operacích, které v C99 nejsou dostupné. Komponenty vektorů jsou označovány dle tabulky na Obr. 12, částečně převzaté z [15].

Počet komponent	Názvy od prvního po poslední člen
2	s0, s1 x, y
4	s0, s1, s2, s3 x, y, z, w
8	s0, s1, s2, s3, s4, s5, s6, s7
16	s0, s1, s2, s3, s4, s5, s6, s7, s8, s9, a, A, b, B, c, C, d, D, e, E, f, F

Obr. 12 Zápis komponent vektoru v OpenCL

Při dodržení tohoto názvosloví je možné psát v kódu složené operace. Následující příklad uvádí, jak výběrově sečíst první dvě komponenty vektoru o 4 položkách.

```
float4 VektorA = (float4) (1.0, 2.0, 3.0, 4.0);
float4 VektorB = (float4) (5.0, 6.0, 7.0, 8.0);
float4 VektorC = (float4) (9.0, 10.0, 11.0, 12.0);

VektorA.xy = VektorB.xy + VektorC.xy
```

Obr. 13 Kód pro výběrové sečtení komponent

OpenCL dále podporuje i uživatelsky definované struktury, které lze využít k vytváření konstrukcí specifických pro řešenou úlohu. Teoreticky je tak možné doplnit vlastní datový typ, např. vektoru o 3 komponentách, který není v jazyku standardně přítomen.

Podporovány jsou i ukazatele. Při jejich používání je však třeba dbát zvýšené pozornosti, jelikož model práce s pamětí je v některých oblastech OpenCL odlišný od toho, na který je programátor aplikací na PC zvyklý, jak již bylo vysvětleno v obecněji pojaté části o programování GPU.

Při deklaraci proměnných a konstant je vhodné uvádět tzv. *address space qualifiers*, které určují, ve kterém z typů paměti, popsaných v kapitole *Programování GPGPU*, budou data uloženy.

Je nutné zdůraznit, že kód v OpenCL C nemůže mít v globálním paměťovém prostoru z kódu deklarovanou proměnnou, v této oblasti lze deklarovat pouze konstanty. Z tohoto pravidla existuje jedna výjimka, která je však dále rozebrána v podkapitole *Kernely*.

5.4.2 Program

Další odlišností od programu v C99 je absence jakékoli funkce *main*, ve které by program začínal a která by představovala jeho hlavní tělo. Jak již bylo zmíněno v části o run time API, program v OpenCL se konceptem spíše podobá dynamicky linkované knihovně. Obsahuje tedy procedury použitelné zvenčí (hostitelem), které nazýváme kernely a pak pomocné procedury, které jsou hostiteli skryty a slouží pouze jako prostředek pro větší modularizaci úloh řešených kernely.

5.4.2.1 Kernely

Kernel je částí programu, kterou lze přímo volat pouze z hostitele. Funkce programu, které jsou využívány jako kernely, je nutné definovat s prefixem *__kernel* a nesmí vracet žádnou návratovou hodnotu. Veškerá data plynou do funkce i ven zpět hostiteli přes parametry kernelu. Parametry mohou být téměř libovolného typu, s výjimkou těch datových typů, které závisí na dostupnosti na dané platformě, jako je např. typ double.

Address space qualifier parametrů může v případě ukazatelů nabývat podoby *__global*, *__local* nebo *__constant*, pro ostatní typy je pak doporučeno použít *__private*.

Obr. 14 ukazuje jednoduchý kernel, sčítající položky dvou polí do třetího, cílového pole. Všechny parametry jsou umístěny v globální paměti, proměnné uvnitř kernelu jsou pak alokovány v oblasti soukromé paměti, není-li určeno jinak.

Odkud se však datové objekty z oblasti globální paměti berou? Jedná se výhradně o paměť alokovanou hostitelem, pomocí dříve zmíněného run-time API.

```
__kernel void vectorAdd(__global const float * a,
                        __global const float * b,
                        __global       float * c)
{
  // Získání ID zpracovávané položky
  int nIndex = get_global_id(0);

  // Součet komponent
  c[nIndex] = a[nIndex] + b[nIndex];
}
```

Obr. 14 Triviální kernel pro součet položek pole

Přestože specifikace OpenCL zmiňuje možnost psát tzv. nativní kernely, běžící přímo na hostitelském hardware, nebude se jimi práce dále zabývat, jelikož nevedou k provádění kódu na GPU a zatím nejsou široce používány.

5.4.2.2 Pomocné funkce

Jistou zvláštností implementací OpenCL na GPU je fakt, že pomocné procedury jsou v současné době implementovány jako tzv. *inline* funkce[2]. To znamená, že ve výsledném kódu nedochází ke skutečnému volání funkcí, ale místo toho je kód funkce expandován v místě volání.

Tento přístup vede v případě kratších funkcí ke zlepšení výkonu, na druhou stranu v současné době není možno toto chování zakázat u větších funkcí, kde již expanze nemusí být žádoucí. To je může způsobovat problémy i u zanořených „volání“, kdy ve výsledku vznikne velký monolitický kód, který může snáze vyčerpat zdroje dostupné na zařízení. Z principu inline funkcí pak přímo vychází další z omezení jazyka

[2] Dle hlášení překladače

oproti standardu C99 a tím je v současné době absence podpory pro ukazatele na funkce.

Pomocné procedury mohou vracet libovolný nativní, či uživatelsky definovaný typ a mohou být volány libovolnou jinou pomocnou funkcí či kernelem.

Z omezení specifických pro OpenCL C je podstatná zejména absence možnosti rekurentních volání funkce. Toto omezení nemusí být závažným nedostatkem, rekurzi lze ve většině případů nahradit nějakou formou cyklu. To však může přinést určitou výkonovou penalizaci, jak je uvedeno dále v podkapitole *Větvení a cykly*.

5.4.2.3 Předdefinované funkce

OpenCL C v současné době nepodporuje žádné standardní hlavičkové soubory, které známe z C, jako jsou *stdio.h*, *string.h* a další. Přesto je však programátorovi k dispozici rozsáhlá sada matematických funkcí, funkcí pro práci s *image objects*, funkcemi pro synchronizaci a konečně funkcemi specifickými pro prostředí paralelních výpočtů. Kompletní seznam a popis funkcí je uveden v [15], přesto zde bude uvedeno několik příkladů pro ilustraci.

Zajímavostí je možnost většinu funkcí vyvolat s vektorovým parametrem. V praxi to znamená, že je možné například funkci *cos(x)* předat vektor o 16 položkách, s tím, že funkce vrátí nový vektor o 16 položkách, který je tvořen kosiny komponent vektoru původního. Podobně lze využívat i funkce jako *max*, *min* a další, což přináší značnou výkonovou úsporu.

Dále lze využívat celé řady poměrně specifických funkcí, jako je *cospi(x)*, představující ekvivalent volání *cos(π, x)*.

K použití je připraveno i několik obměn klasických matematických funkcí, které jsou označeny jako tzv. nativní. Tyto funkce garantují nejrychlejší provedení výpočtu na dané platformě, ovšem s nevýhodou možného snížení přesnosti.

Co se týče přesnosti obecně, všechny implementace OpenCL jsou povinny dodržovat standard IEEE 754[17]. Jednoduchá přesnost je garantována, přesnost dvojitá je pouze výsadou dražších a novějších modelů, jak bylo zmíněno dříve.

V závislosti na implementaci může být implementována funkce *printf*, v případě GPU je však lépe předpokládat její absenci, a využívat ji pouze k ladění kódu, jak je uvedeno v kapitole 8.2.

Velice důležitou skupinou jsou pak funkce, které kernelu dávají informace ohledně nastavení paralelizace, které je kompletně parametrizovatelné pomocí run time API na straně hostitele, a kernely by jinak nemohly získat představu o řešeném problému. OpenCL C pro tyto účely obsahuje několik funkcí, pro ilustraci si představíme tři z nich na modelovém případě zpracování pixelu obrazu v HD rozlišení. Jedná se o klasický případ dvourozměrného problému, což kernel může ověřit voláním funkce *get_work_dim()*, která v tomto případě vrátí hodnotu 2. Pro zjištění šířky obrázku pak intuitivně slouží *get_work_size*, která s parametrem 0 vrátí 1920, a s parametrem 1 pak 1080. Chceme-li zjistit, který pixel aktuálně obsluhujeme, zavoláme v kernelu funkci *get_global_id(0)* pro x-souřadnici a *get_global_id(1)* pro souřadnici y.

5.4.2.4 Větvení a cykly

Programy v OpenCL C mohou využívat všech konstrukcí pro větvení a cykly v kódu známých z C99 – *if*, *switch*, *for*, *while* a dalších. Nelze přehlédnout, že tzv. *dynamic branching*, tedy větvení kódu za běhu, které je na GPU podporováno relativně krátkou dobu, není v současné době implementováno bez postihu na výkon.

Pro zlepšení výkonu je možno pomocí direktivy *#pragma unroll* vynutit rozbalení obsahu smyček cyklu *for*. V případech, kdy nelze předem zjistit počet opakování cyklu,

je kód vykonáván klasicky, tedy často o dost pomaleji, než v případě rozbalené varianty.

6 NÁVRH A REALIZACE KNIHOVEN

Po konzultaci s vedoucím práce byl pro realizaci dynamických knihoven zvolen jazyk C++ s tím, že knihovna bude mít navenek standardní procedurální rozhraní. Toto rozhodnutí vycházelo z požadavku, aby byla knihovna dále použitelná z libovolného programovacího jazyka, což by např. v případě *class library* technologie .Net nebylo zcela splněno. Teoreticky by sice bylo možno pro knihovnu realizovanou v C# vytvořit COM wrapper[18], jednalo by se však o krkolomné řešení, nehledě na vynucenou závislost na run-time knihovnách .Net. Za výhodu volby C++ byla považována okamžitá dostupnost hlavičkových souborů pro OpenCL, i to, že je jazyk v prostředí *Microsoft Visual Studio Express* na škole již zaveden a vyučován.

Z důvodu problémů se statickým linkováním, které Visual C++ mělo ještě na podzim 2009 při integraci s implementací OpenCL s vývojovým kitem firmy Nvidia a z důvodu obtížné přenositelnosti Visual Studia na flash disku, což se ukázalo jako podstatná nevýhoda při vývoji software na různých konfiguracích, byl pro prototypování a testování využíván i jazyk *ThinBASIC*[19], který je dlouhodobým společným projektem autora této práce. Finální knihovny jsou pak kompletně realizovány jako projekt v jazyce PowerBASIC.

Jelikož v době zahájení vývoje aplikací v rámci této diplomové práce bylo OpenCL ještě velmi mladou technologií, u kterých nebývá neobvyklé, že se mění volací konvence, počet i význam implementačně specifických parametrů, bylo rozhodnuto, že knihovna, později poskytující přístup k matematickým výpočtům, bude spoléhat na mezivrstvu nad OpenCL, která bude kompenzovat zmíněné problémy. To znamená, že kód knihovny bude stále stejný, pouze mezivrstva bude v případě změn práce s OpenCL poupravena tak, aby celá knihovna mohla fungovat dále bez zásahu.

Zmíněná mezivrstva bude v dalším textu označována jako *low-level knihovna* (LLK), a jejímu návrhu se bude věnovat zbytek této kapitoly. Nadřazené knihovny jsou pak dále popsány v následujících podkapitolách.

Jelikož knihovna OpenCL není standardní součástí systému Windows, a tudíž se na její přítomnost nebylo možné spoléhat, je v LLK k této knihovně přistupováno metodou pozdní vazby, na platformě Windows zprostředkované pomocí Win32 funkcí *LoadLibrary*[20], *GetProcAddress*[21] a *FreeLibrary*[22]. Přestože AMD i Nvidia nabízejí na míru svému hardware upravené verze hlavičkových souborů OpenCL, v práci byly jako základ využity hlavičkové soubory[23] přímo od fy Khronos, která OpenCL oficiálně zaštiťuje, pro eliminaci platformě specifických závislostí. Deklarace funkcí v těchto souborech pak byly upraveny skriptem do podoby vhodné pro volání pomocí pozdní vazby.

Low level knihovna pak poskytuje následující funkce k dalšímu použití vyššími vrstvami:

- OpenCL_BeginWork
- OpenCL_EndWork
- OpenCL_CreateProgramAndKernel
- OpenCL_DestroyProgramAndKernel
- OpenCL_BuildGPULibrary
- OpenCL_ReleaseGPULibrary
- OpenCL_CreateQueue
- OpenCL_ReleaseQueue
- OpenCL_EnableErrorReport
- OpenCL_DisableErrorReport

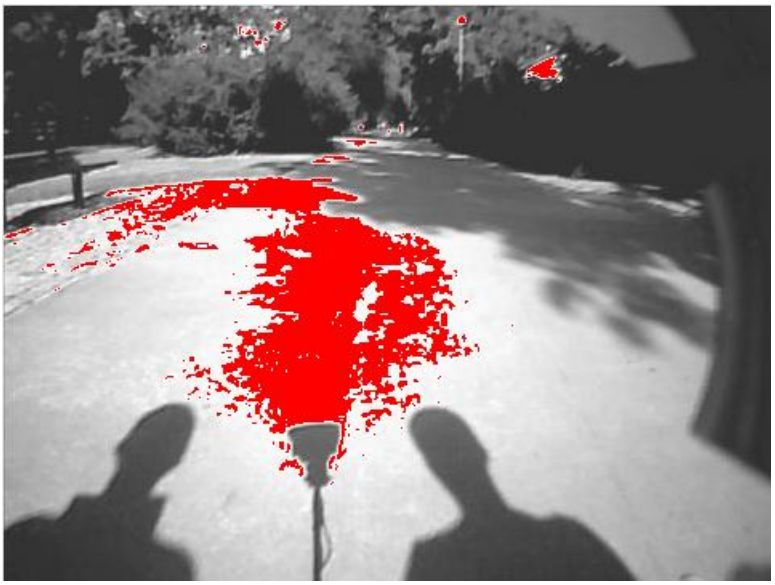
Knihovna tak do značné míry zjednodušuje práci s OpenCL, včetně možnosti automatického hlášení chyb, které jsou okamžitě vypisovány do konzole nebo zapisovány do souboru, takže má programátor přehled o příčinách problému na straně run time OpenCL API.

Vyšší míra abstrakce, umožňující např. samotné spouštění kernelu, už však není možná, což je omezení plynoucí z návrhu run time API samotného OpenCL. I tento problém je však do značné míry řešitelný nástrojem popsaným v kapitole *Generátor kódu*. Programátor je tak schopen vytvářet nové knihovny pro práci s grafickou kartou tak, že projekt LLK, přiložený k této práci na datovém nosiči, použije jako výchozí pro svou další tvorbu s dále popsanou možností využití generátoru kódu.

6.1 Knihovna pro zpracování obrazu

Zpracování obrazových dat je jednou z oblastí využití počítače, se kterou se setkáváme velice často. Pro tyto účely je možno použít běžně dostupné knihovny a software, jedná se však typicky o implementace využívající pro výpočet procesor. Jelikož se v mnoha případech úprava obrazu omezuje na opakování stejné operace pro každý pixel, jedná se o vhodnou úlohu pro data paralelní programování, tedy oblast, ve které vynikají moderní grafické karty.

Realizovaná knihovna tak pokrývá několik problémů z oblasti zpracování obrazu. Jedná se jak o základní operace, jakými je vzájemná záměna barevných komponent obrazu, jeho převod do černobílé podoby, či izolace barevné složky, tak i o některé pokročilejší metody, viz Obr. 15.

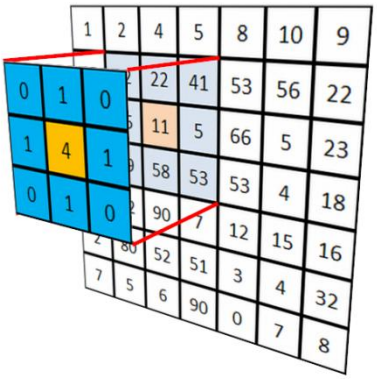


Obr. 15 Indikace přepálených pixelů

6.1.1 Konvoluční filtry

Knihovna dále implementuje i 2D konvoluční filtry[24], které nacházejí široké uplatnění v mnoha oblastech, včetně mobilní robotiky. Základním pojmem z problematiky konvolučních filtrů je tzv. *konvoluční jádro*, které je reprezentováno čtvercovou maticí. Tato matice je posouvána po zpracovávaném obraze tak, že její střed překrývá zpracovávaný pixel a zbylé prvky pak jeho nejbližší okolí. Čísla obsažená v matici jsou koeficienty, kterými jsou hodnoty jednotlivých barevných složek obrazu násobeny. Hodnota barvy pixelu je pak součtem všech hodnot takto upravených, výsledek je obvykle normalizován dělitelem specifickým pro daný problém, případně

upraven dodatečným konstantním posunem, který pouze ke všem hodnotám komponent pixelu přičte stejnou hodnotu.

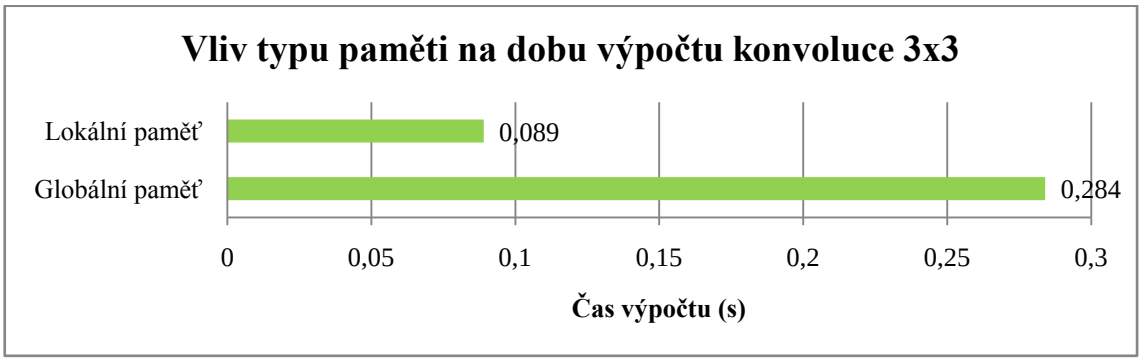


Obr. 16 Konvoluční jádro nad zpracovávanými daty

Několik základních konvolučních filtrů je v knihovně přítomno přímo ve formě procedury. Jedná se např. o gaussovský vyhlazovací filtr typu dolní propusť, zaostřovací filtr typu horní propusť i jednoduchý gradientní filtr pro detekci hran. Pro větší všestrannost knihovny je pak uživateli umožněno definovat filtr vlastní, a to o velikosti 3x3 nebo 5x5 s možností definice dělitele i posunu.

Realizace vlastního filtru v OpenCL přinesla několik výkonnostních problémů, se kterými bylo nutno se vypořádat. Vzhledem k tomu, že knihovna jako taková byla navrhována tak, aby ji bylo možno využívat na hardware dostupném v rámci školy, bylo už od začátku realizace nutné brát v potaz některá omezení. Prvním byla, u grafických karet překvapivá, absence široké podpory pro datový typ *image object*, který by jinak byl nejvhodnější volbou pro reprezentaci dvourozměrné bitmapy. Tento datový typ není podporován na poměrně rozšířených grafických kartách Radeon v sérii HD 4000, a dle vyjádření výrobce tato vlastnost nebude doplněna ani v budoucích ovladačích, z důvodu hardwarového omezení architektury. Na modelech série HD 5000 je již tato funkcionality přítomna, grafické karty tohoto typu však nejsou na škole dostupné.

Jelikož jsou obrázky typicky reprezentovány pomocí tří komponent, každé s 8 bitovou přesností, se zarovnáním v paměti na blok o velikosti 32 bitů, byl jako vhodná reprezentace bitmapy vybrán ukazatel na pole hodnot typu *uchar4*.



Obr. 17 Porovnání vlivu typu použité paměti na Nvidia GeForce 9500GT

Dalším problémem byla zdánlivě nepodstatná otázka umístění uživatelského filtru v paměti GPGPU. Jak již bylo naznačeno v kapitole o OpenCL, v rychlosti přístupu do paměti mohou být v závislosti na jejím typu velké rozdíly. Tento

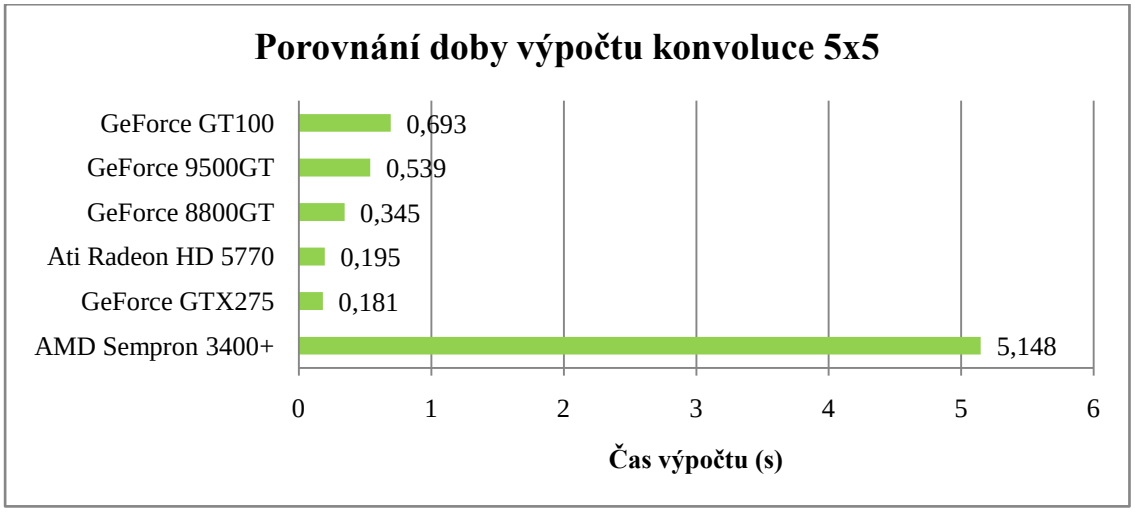
předpoklad se potvrdil, jak ukazuje graf na Obr. 17, kde je jasně patrné, že bylo vhodné toto omezení nějakým způsobem překonat.

Jednou z možností, která se nabízela, bylo umístění uživatelské matice do rychlé lokální paměti. Jak bylo ověřeno, tento krok by přinesl významné zrychlení, ovšem zde narážíme na další problém s dostupným hardware, kterým je pouze částečná podpora hardwarové lokální paměti.

Na zmíněné řadě HD 4000 je tato funkcionalita emulovaná pomocí pomalejší globální paměti. Během testování se však prokázaly jisté problémy s emulací, způsobující při využívání lokální paměti na této platformě při provozu pod Windows 7 náhodné zamrzání aplikace.

Východiskem z této situace bylo využití jedné z výhodných vlastností technologie OpenCL, kterou je dynamická kompilace kódu. Protože matice 3x3 představuje pouhých 9, resp. matice 5x5 25 hodnot, jsou tyto parametry přímo vloženy na určené místo v kódu, do těla kernelu, který je poté zkompilován. Touto jednoduchou úpravou se problémová data náhle objevila v nejrychlejší paměti privátní a uživatelsky definovaná konvoluce tak může dosahovat shodného výkonu se specializovanými konvolučními kernely zmíněnými v úvodu.

Funkce zaměřené na práci s konvolucí tak ve výsledku podávají dobrý výkon i na low end modelech grafických karet, jakým je např. pasivně chlazená GeForce 9500GT.



Obr. 18 Rychlost konvoluce pro obrázek v rozlišení 3840 x 2160

Je třeba zdůraznit, že výhoda GPU oproti procesoru se projevuje výrazněji hlavně pro obrázky ve vyšším rozlišení. Rychlost provádění by bylo možno ještě více zvýšit rozložením jednotlivých bloků obrázku do lokální paměti. Vzhledem ke zmíněným problémům s některými kartami však bylo od tohoto řešení upuštěno. Výsledné realizované řešení, které je v knihovně k dispozici tak je stále výrazně rychlejší než implementace na CPU, s tím, že je možno jej využít na širším spektru grafických karet.

Obr. 19 Obraz z kamery zpracovaný dvouprůchodovým gaussovským filtrem

Pro úplnost pak knihovna obsahuje funkce, které provádí konvoluci efektivnějším dvouprůchodovým způsobem, kdy jsou nejdříve zpracovány řádky, a poté sloupce. Tento způsob je však aplikovatelný pouze u symetrických konvolučních matic, které lze takto rozložit. Implementace je, co se využití možností jazyka OpenCL C týče, provedena podobně, jako v případě celých matic.

6.1.2 Segmentace obrazu

Převedení konvoluce na grafickou kartu přivedlo významné zrychlení, je však třeba podotknout, že se jedná o zrychlení operace již použitelně rychlé i v rámci CPU implementace.

V oblasti zpracování obrazu existují i časově náročnější problémy, jako je segmentace obrazu. Tato třída úloh je hojně využívána v mobilní robotice, např. pro rozpoznávání cesty. V knihovně je tak posledním významnějším implementovaným algoritmem aplikace z této oblasti, konkrétně Mean-Shift algoritmus, detailně popsaný v [25].

Obr. 20 Porovnání původního obrazu a jeho nové podoby po několika iteracích Mean-Shift algoritmu

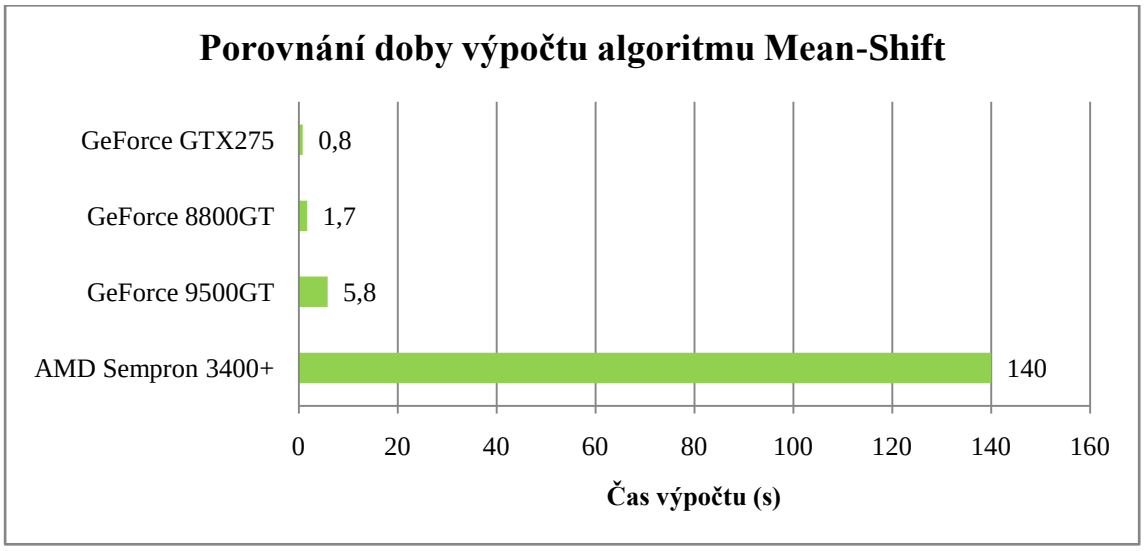
Výkon Mean-Shift nemá tak jednoznačnou charakteristiku jako 2D konvoluce, kde je počet operací jasně definován rozměry obrázku a velikostí matice.

Tento algoritmus se skládá z několika fází. V prvním kroku je z nejbližšího okolí vybráno několik kandidátů pro další zpracování. Zde je možno narazit na první problém. Jejich počet totiž nemusí být dopředu znám, a proto programátora jako první nejspíše napadne použití nějaké formy dynamického pole či seznamu. Jak bylo uvedeno v kapitole o OpenCL, dynamická alokace paměti z kódu kernelu není možná. Proto je

v tomto případě nutné využít statického pole, které je dimenzováno na maximální počet položek, čímž na jedné straně potenciálně plýtváme pamětí, na straně druhé však neprovádíme žádné dynamické alokace, které jsou jinak často zdrojem zpomalení chodu programu.

Po počátečním výběru kandidátů pro zpracování, který prozkoumává jasně definované okolí, přichází druhá fáze, ve které je již počet cyklů předem neznámý a opakování je dosaženo pomocí *while* cyklu. Spolu s dalším zanořeným *for* cyklem je tento případ pro GPU obtížně optimalizovatelnou operaci, nelze například dostatečně efektivně využít rozbalení vnějších smyček. V tomto případě tak lze pouze doporučit rozbalení smyček vnitřních, pomocí direktivy *#pragma unroll*.

Co však již lze optimalizovat poměrně dobře jsou dílčí případy práce s vektory hodnot, pro které má OpenCL zavedenu dříve zmíněnou zjednodušenou syntaxi, která kromě programátorova pohodlí zajišťuje provádění těchto operací pomocí optimalizovaných instrukcí.



Obr. 21 Doba zpracování algoritmu Mean-Shift na GPU i platformě .Net pro obrázek 240x180

I přes zmíněné implementační problémy lze pozorovat na verzi pro GPU velmi výrazné zrychlení, které oproti C# implementaci na moderním dvoujádrovém procesoru přináší až třicetinásobné zrychlení. Jistou daní za složitost algoritmu je však fakt, že na GPU lze tímto způsobem zpracovávat obrazy do velikosti cca 500x400 pixelů, v případě větších rozměrů je již pravděpodobné vyčerpání zdrojů. Uvedená velikost bitmapy však většinou postačuje, používá-li robot uvedené metody pro zpracování obrazu z web kamery. Další teoretickou možností by bylo větší obraz nejdříve zmenšit, a teprve pak zpracovat pomocí realizované GPU implementace.

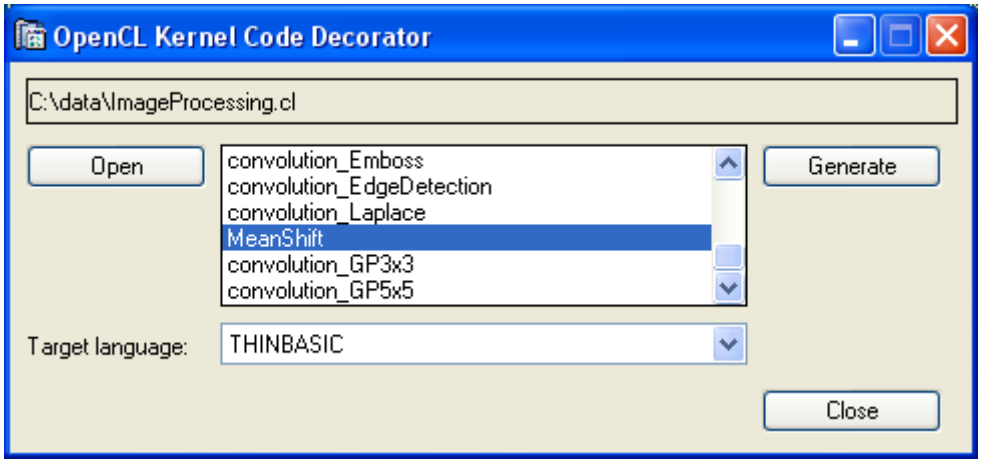
6.2 Generátor kódu

Syntaxe jazyka OpenCL C je velice úsporným způsobem zápisu algoritmů. O run time API, které jednotlivé programy uvádí do chodu, však nelze tvrdit to samé. OpenCL je záměrně navrženo tak, aby ho bylo možné provozovat na různých typech hardware.

Příkazy, které připravují a spouštějí samostatný výpočet, zajišťují zápis, aktualizaci a zpětné získání dat a provádějí celou řadu dalších nezbytných činností, tak díky své obecnosti reálně požadují po programátorovi napsání desítek řádků kódu. Tento požadavek platí i pro případy velice jednoduchých kernelů. Pro každý kernel s odlišnými parametry by tak bylo nutné stále dokola opakovat tutéž repetitivní práci.

Jakákoli opakovaná činnost je velice náchylná k chybám z nepozornosti, a díky své často až ubíjející povaze může mít negativní dopad na motivaci k tvoření nových věcí, v případě OpenCL kernelů.

Do jisté míry je realizace programu prostřednictvím OpenCL usnadněna dříve zmíněnou low level knihovnou, která však z principu nemůže napomoci při přípravě spuštění výpočtu.



Obr. 22 Uživatelské rozhraní generátoru je navrženo pro jednoduchost použití

Z tohoto důvodu byl navržen generátor kódu, který na základě uživatelem vybraného kódu kernelu automaticky vygeneruje 95% kódu potřebného pro jeho spuštění.

Generátor provede analýzu kódu v OpenCL C, ve které jde hlavně o zjištění počtu předávaných parametrů, jejich názvu, datového typu a také informace, jedná-li se o parametr konstatní, určený k zápisu či čtení. Na základě toho jsou pak poměrně přesně vytvořeny příslušné definice paměťových objektů, které OpenCL API ke komunikaci s kernelem vyžaduje.

Další analyzovanou položkou je pak počet rozměrů problému, aby mohlo být korektně vygenerováno volání pro spuštění kernelu.

Na programátorovi samotném je potom už specifikace zdroje dat a objemu výpočtu. I tak však může tento pomocný nástroj významně ušetřit čas strávený vývojem OpenCL aplikací.

7 PRAKTICKÁ APLIKACE OPENCL

Řešení předvedená v minulé kapitole byla ukázkou využití OpenCL pro obecně známé problémy. V této kapitole bude popsána konkrétní praktická aplikace OpenCL, jejímž výsledkem bude řádové zkrácení doby výpočtu[26].

7.1 Metoda PCSM

PreComputed Scan Matching je metoda lokalizace robotu v interiérech budov, původně vyvinutá Ing. Stanislavem Věchetem, Ph.D. na VUT v Brně. Tento postup je určen pro roboty vybavené laserovým dálkoměrem, tedy stroje, schopné získávat informaci o okolí na základě vějíře paprsků odrážejících se od nejbližšího okolí. PCSM je rozděleno na dvě fáze – přípravnou fázi a samotnou lokalizaci.

Lokalizace pomocí této metody je velice rychlá, a její optimalizovaná verze je popsána v literatuře[27]. Zjednodušeně řečeno, během lokalizace dochází k porovnávání vstupu ze senzorů s daty z přípravné fáze, jak ukazuje názorné video v příloze C.

Dosavadní nevýhodou PCSM tak byly hlavně značné časové nároky zmiňované fáze přípravné, kdy dochází k předpočítání kolize 360° růžice paprsků s modelem prostředí. Doba trvání výpočtu závisí do značné míry na velikosti prostředí, kde se bude robot pohybovat, a požadované přesnosti. Přesnost lokalizace vychází ze stupně pokrytí plochy místnosti růžicemi. S rostoucí hustotou přesnost roste.

7.2 Motivace k optimalizaci pomocí GPGPU

V případě rozsáhlého prostoru, jakým je např. přízemí budovy A1 Fakulty strojního inženýrství, je i pro relativně hrubou přesnost 20 cm třeba spočítat kolize 360 paprsků se všemi překážkami v desetitisících odlišných bodů. Tento v podstatě triviální výpočet svým objemem způsobuje, že i na moderním procesoru může trvat i několik minut. Taková implementace pak tedy vyhovuje pouze příležitostnému offline spuštění, a pozdějšímu přenesení dat na robot. V situaci, kdy se oblast působení robotu může měnit, je pak robot nucen čekat na nová data od operátora.

Představme si ovšem rizikovou situaci, kdy se robot pohybuje v měnícím se prostředí a kdy by bylo vhodné, aby přepočet kolizí pro PCSM byl proveden přímo na robotu, pouze na základě dodané aktualizované mapy, kterou mu operátor-pozorovatel zašle. V takovém případě není možné čekat několik minut a následně přenášet, byť bezdrátově, velký objem předpočítaných dat na robot.

Nejen pro použití v těchto situacích je pak tedy vhodné výpočet urychlit.

7.3 Paralelizace PCSM z pohledu OpenCL

Provádění stejné úlohy, v tomto případě kolize paprsků s okolím, v různých bodech prostoru je zřejmě jednoduše paralelizovatelnou úlohou. GPGPU s velkým počtem *processing elements* se tak mohou zdát ideální volbou přístupu k řešení problému. Při realizaci je však třeba mít na paměti i různá omezení tohoto typu hardware.

Nejčastějšími překážkami, se kterými se programátor na současném hardware může setkat je:

- Nedostatečně rozšířená podpora double precision
- Značný dopad větvení kódu na výkon
- Omezení velikosti parametrů pro kernel
- Vyčerpání zdrojů pro *processing element*

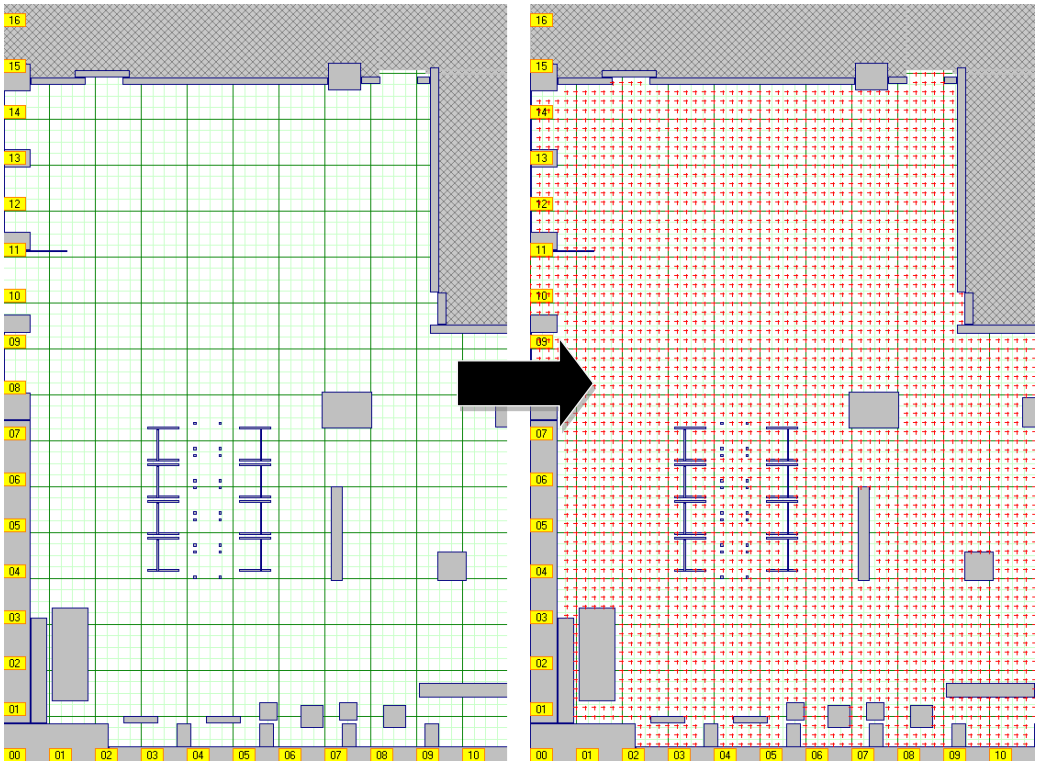
První nedostatek lze obejít vhodnou volbou jednotek pro výpočet. Jelikož je PCSM určeno pro interiéry budov, lze předpokládat, že prostory budou relativně malé.

Pro potřeby tohoto výpočtu tak bude postačovat jednoduchá přesnost či dokonce celočíselné typy, zvolíme-li jako jednotky milimetry.

Problém s redukovaným výkonem při větvení vyplývá z faktu, že grafické karty dlouhou dobu nepodporovaly dynamické větvení kódu vůbec. Z tohoto důvodu je vhodné omezit počet podmínek a větvení v kódu. Aby k tomuto negativnímu jevu nedocházelo, je předpočet dat pro PCSM rozdělen na dva kernely.

7.3.1 Hledání bodů pro kolizi

V první fázi je nutné najít body, ve kterých bude kolize paprsků s prostředím provedena. Jediným úkolem kernelu tak bude vyhledání bodů, které jsou mimo tělo překážek a představují tedy možnou oblast působnosti robotu, jak ukazuje Obr. 23.



Obr. 23: Vlevo půdorys budovy před výpočtem, vpravo s nalezenými body výskytu robotu

Již při realizaci této operace narazíme na problém související s jednou z komplikací zmiňovaných v úvodu a tou je možné vyčerpání zdrojů. Příčina tohoto problému byla nejdříve autorem práce mylně vyhodnocena jako překročení maximální velikosti parametru, který kernel může přijmout, což po určitou dobu vedlo cesty řešení problému špatným směrem. Jak již bylo zmíněno, skutečný problém spočíval ve vyčerpání zdrojů, ovšem ne při předání parametru, ale až v průběhu samotného výpočtu.

Toto je jeden z negativních důsledků problematické implementace dynamického větvení a opakování na většině dostupných GPU současnosti. Velikost dat, která nesla informaci o překážkách, pak byla až sekundárním faktorem, který vedl k rychlejšímu vyvolání problému. Překážky byly reprezentovány jako čtverec, popsaný pozicí, rozměry a viditelností. V průběhu realizace pak došlo ke zcela mylnému vyvození souvislosti mezi maximální velikostí parametru, a zpracovatelným objemem překážek. Odhalení pravé podstaty problému bylo dosaženo až při testování metody na jiném typu GPU, kde již žádná souvislost mezi maximální velikostí parametru a zpracovatelným objemem překážek nebyla tak jednoznačná.

V konečném důsledku však jak původní mylný předpoklad, tak skutečná odhalená příčina vedla ke stejnému řešení. Jelikož reálné prostory obsahují velké množství překážek, které by grafická karta díky zmíněným problémům nebyla schopna zpracovat zároveň, došlo k rozložení výpočtu na několik průchodů.

Toto řešení však potenciálně hrozilo zvýšeným provozem na sběrnici mezi hostitelem a zařízením, což by mělo velice negativní dopad na výkon. OpenCL však nabízí elegantní řešení, které toto omezení umožňuje do značné míry minimalizovat. Tímto řešením je možnost aktualizovat pouze vybrané parametry kernelu, v tomto případě informace o překážkách, a zbylé parametry, jako je pole pro vyplnění souřadnic bodů dodat pouze jednou, na začátku výpočtu. Je třeba zdůraznit, že toto rozdělení není ekvivalentním s dělením na pracovní skupiny, popsaném v kapitole věnované OpenCL. Dělení problému je v tomto případě plně v rukou programátora, v případě nevhodného rozvržení problému, které by nebralo v potaz omezení hardware, lze očekávat chybové hlášení *cl_out_of_resources*.

Zmíněné body, jejichž přítomnost v překážce testujeme, jsou pak reprezentovány jako trojice čísel. První dvě čísla určují souřadnici bodu, poslední číslo je pak jen příznakem, který určuje, zda se bod nachází v překážce či nikoli. V případě, že je bod v libovolném průchodu výpočtu nalezen uvnitř překážky, je příznak nastaven na hodnotu 1, jinak zůstává na výchozí nulové hodnotě.

7.3.2 Provedení výpočtu kolize paprsků s modelem

Druhý kernel je určen k provedení kolize paprsků s prostředím v bodech, které byly v předchozím kroku označené, jako potenciální působiště robotu. I v tomto případě je problematickým místem algoritmu zpracování informace o všech překážkách, což je opět řešeno více průchody. Jelikož byla tato operace vysvětlena pro předchozí pomocný výpočet, nebude zde znovu popsána, a na místo toho uvedeme jiné specifikum tohoto kernelu.

Protože je kolize paprsků složitějším problémem, než přítomnost bodu v pravoúhlé oblasti, je vhodné rozdělit si řešení na dílčí celky. Jak bylo zmíněno v kapitole o OpenCL, výpočet zpracovávaný na GPU má podobu kernelu, který může využívat pomocných procedur. V tomto případě byl použit jeden hlavní kernel, který výpočet dále rozkládal na dvě interní procedury:

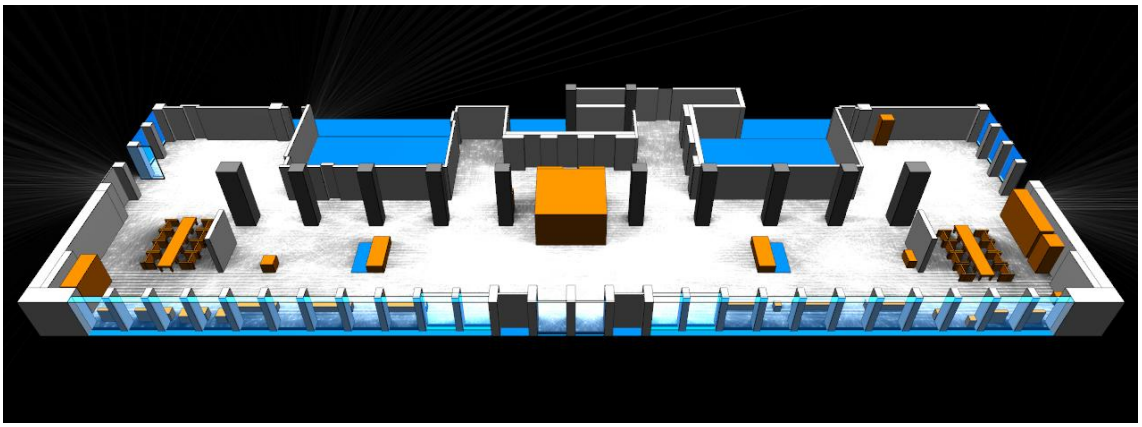
- *Aux_CrossDistance*
- *Aux_RayRectangleDistance*

První procedura provádí kolizi paprsku a úsečky. V případě, že dojde ke kolizi, vrací procedura kladné číslo, které je vzdáleností počátku paprsku k průsečíku s jinou úsečkou. V případě, že průsečík není nalezen, vrací procedura zápornou hodnotu a její výsledek není brán v potaz.

Druhá procedura pak provádí obecně kolizi paprsku s překážkou definovanou jako obdélník. Ten je na GPU rozložen na jednotlivé úsečky, a kolize paprsku je pak za pomocí procedury *Aux_CrossDistance* prováděna pro každou z nich.

Zmíněná druhá procedura je přímo vyvolána z těla kernelu, ve kterém dochází k vytváření 360 paprsků ke kolizi. Cyklus pro tvorbu paprsků pomocí direktivy *#pragma unroll* rozbalen do podoby kódu bez instrukcí opakování, což na jednu stranu urychluje výpočet, na stranu druhou pak činí kód kernelu delším, čímž dochází k většímu vytížení dostupných zdrojů.

Výsledné pokrytí místnosti paprsky je znázorněno na Obr. 24.

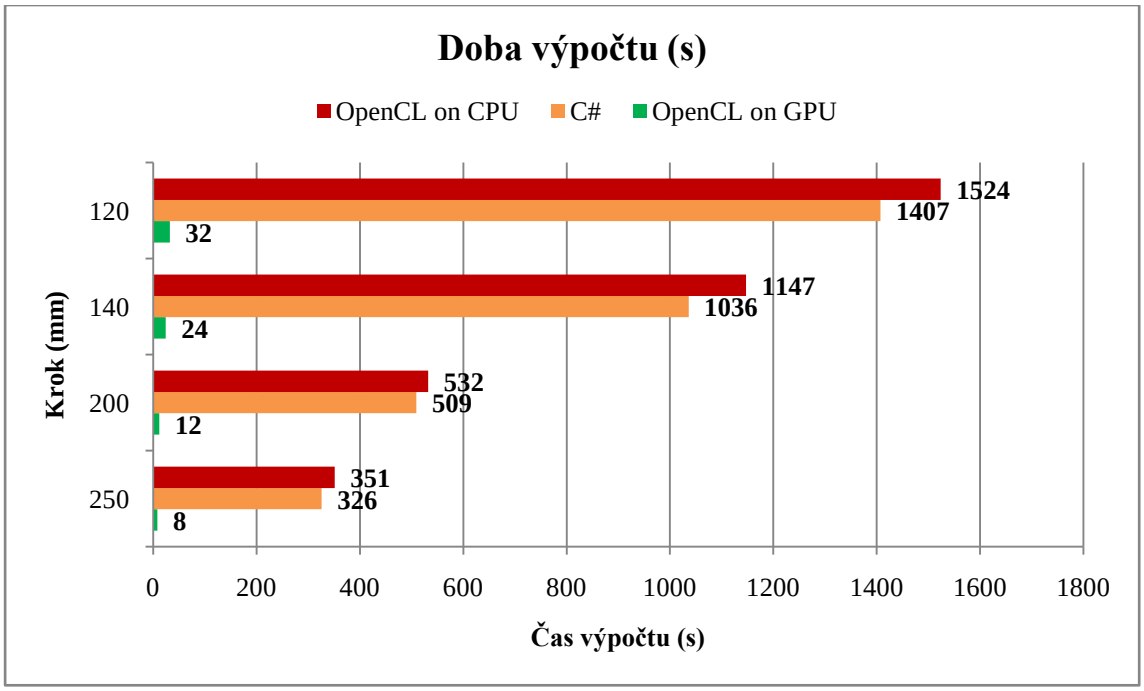


Obr. 24 Vizualizace rozložení paprsků v budově A1

7.4 Přínos optimalizace

Zvolený přístup umožňuje zachovat kód modulární a udržovatelný, zároveň je tato podoba jednoduše rozložitelná na jednotlivé *processing elements*, což vede k značnému zrychlení operace.

Pro ilustraci je na Obr. 25 je uvedeno srovnání výkonu implementace algoritmu na CPU a GPU. Za pozornost stojí horší výkon OpenCL v CPU módu oproti verzi v jazyce C#, toto chování však platí hlavně pro jednojádrové procesory. Časy GPU verze však jasně ukazují výhodu paralelizovaného přístupu na grafickém hardware.



Obr. 25 Porovnání výkonu CPU a GPU implementace

Jak je z grafu patrné, zrychlení PCSM na GPU je oproti verzi pro CPU značné. Výpočet, který na procesoru pro případ přesnosti na 12cm trval téměř půl hodiny byl po převedení na GPU zrychlen do řádu sekund, což zcela mění možnosti praktického použití algoritmu z offline předpočtu na výpočet, který lze na robotu osazeném vhodnou grafickou kartou provést přímo.

8 PRAKTICKÉ ZKUŠENOSTI

Od prvních implementací OpenCL, které umožňovaly chod programů na GPGPU, bylo možno v této oblasti sledovat velice intenzivní vývoj. Během využívání této technologie bylo autorem práce objeveno několik důležitých poznatků a skutečností.

Většina z nich je popsána v samotném závěru práce, přesto však bude v této kapitole popsáno několik praktických pozorování, která shrnují získané poznatky, které se obecně týkají vývoje aplikací nad technologií OpenCL, a tedy i problémů, se kterými se budou programátoři po nějakou dobu potýkat.

8.1 Možné příčiny selhání kompilace

První třída problémů, na které lze narazit na samotném počátku vývoje, je odhalování syntaktických chyb programu. V současných překladačích i interpretech programovacích jazyků si již programátoři zvykli na komfortní chybová hlášení, která oznamují typ chyby i přibližnou polohu řádku, na kterém k chybě došlo.

Tento typ zpětné vazby nechybí ani u současných JIT kompilátorů OpenCL C, kdy se ve většině případů autor programu dočká vysvětlení, proč kód nelze přeložit. Vzhledem k prudkému vývoji oblasti, a možná i díky snaze přivést implementaci technologie jako první, však i dnes, v oblast překladu programu lze narazit na situace, kdy funkce *clBuildProgram* oznámí pouze selhání kompilace, zcela bez udání důvodu.

Hledání chyby je v takovýchto případech velice zdlouhavé, a proto autor práce odeslal technické podpoře firem ATi i Nvidia několik hlášení o problémech s překladem. Navzdory této snaze se i s posledními verzemi OpenCL knihoven stále objevují případy, kdy se program nepřeloží, a neoznámí důvod.

Z tohoto důvodu je zde uvedeno několik typických problémových případů, které vedou ke zmíněné komplikaci, se kterou se v současné době programátoři potýkají. Je s podivem, že většina popsaných problémů je společná jak pro implementace firmy ATi, tak Nvidie.

Jak uvádí [15] v na první pohled přehledné tabulce, pro zápis komponent vektoru je možné použít buď písmen, nebo číslic. Programátor tak snadno nabude dojmu, že první dva zápisy součtu komponent vektoru, jak je uvedeno na Obr. 26, jsou ekvivalentní.

```
// Zapis za pouziti pismen
Vec.xyz = VecA.xyz + VecB.xyz;

// Zapis za pouziti cislic
Vec.012 = VecA.012 + VecB.012;

// Spravny zapis za pouziti cislic
Vec.s012 = VecA.s012 + VecB.s012;
```

Obr. 26 Problémová vektorová notace

Druhý zmíněný zápis však vede k selhání bez chybového hlášení. Důvod je prostý, specifikace mimo tabulku, dále v textu uvádí, že v případě číselných koeficientů je nutné dát jako první znak za tečku písmeno *s*. Tento drobný detail je ovšem často, zvláště ve stresu, velice obtížné odhalit. I z tohoto důvodu je na Obr. 12 v kapitole *Technologie OpenCL* uvedena tabulka podobná té ve specifikaci, ovšem se všemi čísly s prefixem *s*.

Dalším typickým problémem, který vede ke zdánlivě nevysvětlitelnému selhání, je pokus o kombinovaný zápis označení komponent vektoru, tedy čísly i písmeny. Specifikace toto sice explicitně zakazuje, kompilátor tento problém někdy korektně identifikuje, v některých případech však dojde pouze k oznámení o selhání. Tento problém byl častou příčinou problémů při dříve popsané implementaci Mean-Shift algoritmu.

Posledním problémem, který zde bude zmíněn, je opět chyba vzniklá nepozorností. V případě, že je v průběhu vývoje programu změněn typ proměnné z ukazatele na klasickou numerickou proměnnou, jakýkoli pokus o její (chybné) dereferencování zůstane ze strany překladače opět bez konkrétního chybového hlášení.

Nutno dodat, že ve všech případech je případné selhání programu správnou reakcí, nestane se tedy, že by kompilátor akceptoval nesmyslný kód, pouze však v několika případech odmítne oznámit, z jakého důvodu překlad selhal, což značně znesnadňuje hledání chyby.

Situace se v této oblasti významně zlepšuje s každou novou vydanou verzí implementace, přesto autor doporučuje v případě nalezení podobné anomálie odeslat oznámení o chybě přímo vývojáři implementace, čímž by se situace v této oblasti měla výrazně zlepšovat.

Za tímto účelem jsou dostupné internetové stránky, kde po nutné registraci programátor dostane přístup ke specializovanému nástroji pro hlášení chyb [28][29].

8.2 Ladění kódu

Otázka ladění kódu pro optimální výkon je v případě vývoje pro GPU klíčová. V průběhu realizace řešení zmíněných v této práci, kdy vývoj probíhal zejména na platformě Nvidia, tak bylo provedeno několik pokusů o využití dodávaných nástrojů.

Ve většině případů však profilovací programy vykazovaly nestabilní chování, což lze přičíst i faktu, že se většinou jednalo o beta verze. Zatímco nástroje pro ladění CUDA programů fungují velice spolehlivě, v případě jejich alternativ pro rozhraní OpenCL bylo možno setkat se spíše s problémy. Autor práce odeslal hlášení o zjištěných potížích pomocí zmíněných nástrojů pro oznámení chyb, ale zatím nebylo z druhé strany potvrzeno, podobně jako v případě problémů s překladem, zda bude problém v dohledné době nějakým způsobem vyřešen.

Vzhledem k malému rozsahu kódu kernelů tak bylo možno ladit kód z jistého pohledu zastaralým způsobem, kdy jsou před a za kritické řádky vloženy výpisy na standardní výstup. Problémem však v tomto případě byla absence podpory funkce *prinft* na platformě Nvidia. Tento fakt je poněkud zarážející, už z toho důvodu, že Nvidia CUDA obsahuje analogickou funkci *cuPrintf*, přičemž OpenCL je na kartách této firmy navrženo jako nadstavba nad touto technologií.

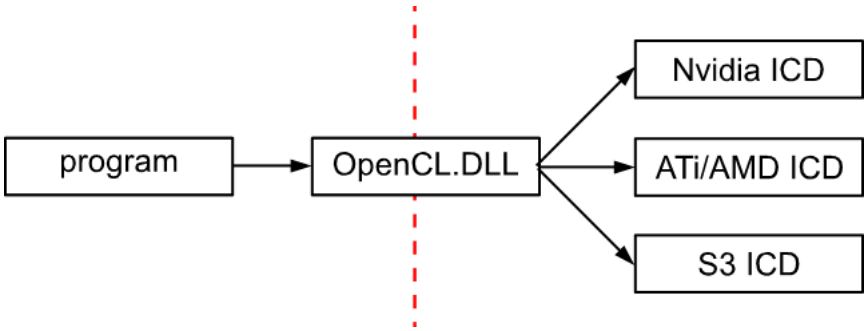
Řešením tohoto problému tak bylo paradoxně využití dynamické knihovny z konkurenčního ATi Stream. Tímto poněkud komplikovaným řešením tak bylo možné ladit problémy v kódu pomocí CPU emulace na procesoru od AMD.

Tento přístup však měl výhodu v možnosti souběžného porovnání implementace OpenCL od fy Nvidia i ATi. Během tohoto procesu se potvrdily některé další problémy, které autor pozoroval již při práci s grafickou knihovnou OpenGL. Firma Nvidia totiž často povoluje používání technik nad rámec specifikace, což na jednu stranu vede k využití plného potenciálu jejich hardware, na straně druhé pak i k tvorbě programů, které na jiných platformách vykazují problematické chování, nebo se vůbec nespustí.

8.3 Distribuce aplikací

Jistý problém nastává v případě, že vlastní řešení na platformě OpenCL chceme distribuovat jako aplikaci. Zde je klíčovou překážkou fakt, že tato technologie, ač

navržena jako knihovna pro práci s heterogenním hardware, stále tímto způsobem nefunguje. Toto je dáno situací, kdy operační systém v současné době neobsahuje jednu jedinou knihovnu OpenCL.DLL, která by dále přesměrovávala na konkrétní pomocné knihovny specifické pro daný hardware, někdy označované jako ICD, tedy *Installable Client Driver*. Tento systém je naznačen na Obr. 27.



Obr. 27 Schematické znázornění hierarchie ICD

Stav je tak nyní zcela odlišný od původní ideje. V případě firmy Nvidia poslední verze jejich ovladačů instalují knihovnu s názvem OpenCL.DLL přímo do systémové složky Windows, což na jednu stranu usnadňuje programování za použití OpenCL na grafických kartách fy Nvidia, na straně druhé však znemožňuje využití modelu naznačeném na Obr. 27.

V případě firmy ATi/AMD je situace zcela odlišná. V tomto případě se žádná knihovna neinstaluje s ovladači vůbec, a pro její získání je nutné instalovat kompletní ATi Stream SDK. Jeho součástí je pak knihovna *atiocl.dll*, která se nijak do systému neintegruje, ale plní stejnou funkci jako *OpenCL.DLL* v případě firmy Nvidia. Přestože byla v různých neoficiálních zdrojích uvedena i informace, že tato knihovna od ATi umožňuje mimo jiné využití i GPU fy Nvidia a procesorů Intel, tato skutečnost se s posledními verzemi SDK při realizaci této práce nepotvrdila.

Další drobnou komplikací jsou pak 2 verze knihovny *atiocl.dll* – 32 bitová a 64 bitová. Aplikace psaná jako 32 bitová je schopná na 32 bitovém systému spolupracovat s 32 bitovou knihovnou, na 64 bitovém systému, pravděpodobně díky 64 bitovému ovladači grafické karty, je nutné pracovat s 64 bitovou verzí DLL.

Jak tedy řešit zmiňované problémy v případě, že chceme distribuovat vlastní aplikaci založenou na OpenCL na libovolné PC pod Windows? Dokud se situace nezmění, a OpenCL nebude do operačního systému integrováno nativně, je dle autora práce nutné pro vlastní aplikaci připravit tzv. chytrý instalátor. Ten bude muset v průběhu instalace detekovat platformu.

V případě, že je OpenCL.DLL již přítomna v systémové složce, není nutné vykonávat žádné další speciální akce. V opačném případě je tak nutné předpokládat, že se jedná o platformu ATi. Instalátor tak musí mít připraveny dvě verze atiocl.dll, které v závislosti na typu operačního systému instaluje do složky programu v příslušné 32 bitové či 64 bitové verzi, s tím, že DLL budou přejmenována na *OpenCL.DLL*.

Pro vyloučení problémů v budoucnu je v případě programů, které si DLL nesou ssebou, vhodné při jejich startu detekovat, zda uživatel později nedoinstaloval novější verzi OpenCL.DLL do systémové složky. V takovém případě je pak možné starou knihovnu smazat, a využívat tak knihovny nové.

Samotné rozhodování ohledně výběru knihovny při instalaci tak může provádět pomocný program. Tato technika je proveditelná i v případě freeware instalátorů, jakým je např. *InnoSetup*[30], který umožňuje jednoduše programovat průběh instalace.

9 ZÁVĚR

Tato práce prozkoumala přístupy k programování grafického hardware, a popsala vývoj v této oblasti od úplných počátků (viz kap. 4.1) až po současnost. Moderní grafické karty jsou dnes, po letech bouřlivého vývoje, vhodnou platformou zejména pro oblast data paralelních výpočtů, ve které svým vysokým výkonem, způsobeným z velké části velkým počtem jader, předčí i nejmodernější vícejádrové procesory.

Jak uvádí kapitola 4.2, v současné době existuje v této oblasti několik řešení. Některá z nich jsou úzce vázána na hardware konkrétní firmy (ATi Stream a Nvidia CUDA), některá jsou navržena pro grafické karty obecně (DirectCompute), s vazbou na určitou verzi operačního systému. Nakonec je zde nové řešení, které nabízí možnost využití všech výpočtu schopných zařízení v PC. Tímto řešením je technologie OpenCL, která je po teoretické stránce podrobně analyzována v kapitole 5, včetně poznámek k několika v praxi zjištěným problémům.

Díky faktu, že první implementace této technologie byly dostupné až v druhé polovině roku 2009, nelze se divit, že má tento přístup i v současné době několik problémů. Za jednu z příčin obtíží vývoje pod OpenCL lze považovat i fakt, že současné implementace získaly certifikaci správnosti od zastřešující společnosti Khronos i přes to, že se jejich chování v několika oblastech rozchází s oficiální specifikací. Některé z problémů se vážou přímo na hardware vybraných firem.

V případě firmy ATi, jejíž výpočetní API procházelo již od počátku velice intenzivním vývojem (viz kap. 4.2.2), představuje problém hlavně několik omezení, která vyplývají, dle názoru autora práce, hlavně z dlouhodobé úzké orientace firmy na herní a grafický průmysl, čemuž se ovšem v případě karet, které označujeme jako *grafické,* nelze příliš divit. Z tohoto důvodu jsou některé klíčové vlastnosti OpenCL, jako je podpora datových typů a operací vázaných na práci s obrazem či přítomnost lokální paměti, na starším hardware nedostupné.

Co však lze považovat za jednoznačně pozitivní fakt, je skutečnost, že se tato firma v posledních měsících velice intenzivně věnuje výrobě výukových materiálů a vzorových řešení pro OpenCL, přičemž průběžně dochází k doplňování některých dosud nedostupných funkcí, jakými je např. kooperace mezi OpenGL a OpenCL. Dalším významným přínosem je pak implementace OpenCL i pro procesory AMD, v jejichž případě dochází k ideálnímu vytížení všech jader.

Z pohledu programátora pak grafické karty poslední řady Radeon HD 5000, dle vyjádření firmy již plně optimalizované pro provoz pod OpenCL, představují velice perspektivní platformu pro provádění výpočtů. Jistým problémem pak může být, v případě některých aplikací, absence dostatečně rychlé implementace operací v dvojité přesnosti.

Vývoj technologie OpenCL na platformě Nvidia se zpočátku potýkal se značnými problémy ohledně standardizace rozhraní knihovny OpenCL (viz kap. 6), což způsobovalo komplikace s vyhotovením této práce. V současné době je však již situace stabilní, a implementace OpenCL je součástí oficiálních stabilních verzí ovladačů. Za výhodu je možno považovat fakt, že oproti platformě ATi, kde lze pro OpenCL využít pouze poslední 2 generace karet, v případě fy Nvidia se jedná o 4 generace hardware. Z tohoto důvodu lze využívat k výpočtům i dnes historickou GeForce 8800, jak prokazuje její účast v některých testech výkonu, uvedených v práci.

Přestože firma S3 oznámila v průběhu roku 2009 kartu schopnou provádění výpočtů přes OpenCL, zatím se neobjevila na českých internetových obchodech a ani její recenze na zahraničních webech nejsou v době psaní této práce dostupné, proto nejsou specifika tohoto řešení uvedeny.

Dalším problémem, který lze s OpenCL pozorovat, je dosavadní absence *Installable Client Driver* modelu, což zabraňuje proklamovanému plnému využití heterogenního hardware i bezpečné možnosti statického linkování na knihovnu OpenCL. Tento problém je však řešitelný přístupy zmíněnými v kap. 6 a 8.3.

Samotná realizace vlastních programů na platformě OpenCL se v této práci ukázala být poměrně problematickou, a to z několika důvodů. Přestože lze narazit na několik návodů a vzorových řešení, jak s OpenCL pracovat, mnoho z nich pochází z doby, kdy ještě nebyl schválen finální návrh této technologie. Není tedy výjimkou setkat se s výpisy kódu, které používají funkce, které v současné verzi standardu OpenCLnejsou přítomny, či zápisy funkčních volání se zcela odlišnými parametry.

Drobnou komplikaci pak představovaly i zmíněné problémy s Nvidia SDK. V únoru 2010 došlo jak na straně ATi tak Nvidie ke změně způsobu inicializace OpenCL programu.

I přes zmíněné problémy, a možná i díky nim, považuje autor za přínosy této práce následující dosažené výsledky:

- Realizace knihovny pro práci s obrazem
- Výrazné urychlení metody PCSM
- Realizace generátoru kódu pro rychlý návrh GPGPU aplikací
- Popsání současných problémů s praktickým využitím OpenCL a návrh jejich řešení

Jak ukazuje kapitola 6.1.1, realizovaná knihovna pro zpracování obrazu umožňuje v porovnání s implementací na procesoru výrazné zrychlení konvolučních operací, řádově 25x a to stále při využití levného GPU, které je představitelem nižší střední třídy. Ještě výraznější je pak přínos GPU verze segmentace obrazu algoritmem Mean-Shift, v jehož případě bylo pozorované zrychlení oproti kódu na platformě .Net osmdesátinásobné a to i za použití nejstarší řady GPU, která je ještě použitelná pro výpočty.

Za velký přínos pak autor práce považuje urychlení lokalizační metody vyvinuté na VUT (viz kap. 7), kdy došlo k řádovému zkrácení výpočtu z desítek minut na desítky sekund. Tato modifikace tak zcela změnila způsob využití metody, kdy již nadále není nutné provádět předpočet dat na vyhrazeném výkonném PC, ale mobilní robot na platformě Nvidia ION je schopen výpočet realizovat v krátkém čase sám. V průběhu realizace bylo ověřeno, že výpočet metody PCSM lze realizovat i na levném GPU, které svým výkonem pro daný problém výrazně předčí i moderní čtyřjádrové procesory. Díky tomuto zjištění lze využít realizovaná řešení i na vybraných platformách snadno využitelných v oblasti mobilní robotiky, jakou je např. Nvidia ION, ať už z důvodu urychlení operace, nebo odlehčení vytížení procesoru.

Dle autora práce je tak již nyní možno považovat technologii OpenCL za dostupný a použitelný prostředek pro zrychlování kritických částí aplikací. Nezanedbatelnou výhodu OpenCL pak představuje možnost psát kód určený k optimalizaci pomocí vysokoúrovňového jazyka, což ostře kontrastuje s optimalizacemi pomocí assembleru. K urychlení vývoje aplikací na této platformě pak slouží nástroj v kap. 6.2.

SEZNAM POUŽITÉ LITERATURY

[1]**Spitzer, John.** GeForce 256 Register Combiners. *Nvidia developer zone.* [Online] 2004.
Dostupné z <http://developer.nvidia.com/object/registercombiners.html>.

[2]**Engel, Wolfgang.** High Level View on Pixel Shader Programming. *GameDev.net.* [Online]
Dostupné z
<http://www.gamedev.net/columns/hardcore/dxshader3/page3.asp>.

[3]**Khason, Tamir.** Technology blog about new Microsoft technologies. E.g. WPF, crossbow, .NET framework 3, Live etc. It includes code sources and samples. *HLSL (Pixel shader) effects tutorial.* [Online] 2008. Dostupné z <http://blogs.microsoft.co.il/blogs/tamir/archive/2008/06/17/hlsl-pixel-shader-effects-tutorial.aspx>.

[4]**I., Buck, a další.** Computer Graphics at Standford University. *Brook for GPUs: Stream Computing on Graphics Hardware.* [Online] 2004. Dostupné z <http://graphics.stanford.edu/papers/brookgpu/>.

[5]**Nvidia.** NVIDIA GPU Computing Developer Home Page. *NVIDIA Developer Web Site.* [Online] 2010. Dostupné z
< http://developer.nvidia.com/object/gpucomputing.html>.

[6]**Group, The Portland.** Portland Group. *PGI | Resources | CUDA Fortran.* [Online] 2010. Dostupné z
<http://www.pgroup.com/resources/cudafortran.htm>.

[7]**PRNewswire.** Just Cause 2 Adds Support for Latest NVIDIA Technologies. *PR Newswire: press release distribution, targeting, monitoring and marketing.* [Online] 2010. Dostupné z <http://www.prnewswire.com/news-releases/just-cause-2-adds-support-for-latest-nvidia-technologies-85999182.html>.

[8]**NVIDIA.** NVIDIA Collaborates With Weta to Accelerate Visual Effects for Avatar. *Welcome to NVIDIA - World Leader in Visual Computing Technologies.* [Online] 2010. Dostupné z
<http://www.nvidia.com/object/wetadigital_avatar.html>.

[9] **Theoretical and Computational Biophysics Group**.NAMD - Scalable Molecular Dynamics. [Online] 2010. Dostupné z
<http://www.ks.uiuc.edu/Research/namd/>.

[10]**Nvidia.** CUDA and GPU Computing Books. *CUDA Zone.* [Online] 2010. Dostupné z <http://www.nvidia.com/object/cuda_books.html>.

[11]**Kolektiv autorů.** GPU Gems 3 - 3D and General Programming Techniques for GPUs. *NVIDIA Developer Web Site.* [Online] 2009. Dostupné z <http://developer.nvidia.com/object/gpu-gems-3.html#tablecontent>.

[12]**Advanced Micro Devices, Inc.** ATi Stream Profiler | AMD Developer Central. *Welcome to AMD Developer Central.* [Online] 2010. Dostupné z <http://developer.amd.com/gpu/StreamProfiler/Pages/default.aspx>.

[13]**Spille, Carsten.** DirectX 11 Compute Shader: Three times faster than DX10.1 due to Local Data Share. *PC Games Hardware.* [Online] 2009. Dostupné z <http://www.pcgameshardware.com/aid,689924/DirectX-11-Compute-Shader-Three-times-faster-than-DX101-due-to-Local-Data-Share/News/>.

[14] **OpenVidia**, Direct Compute Shader Code Listing.. [Online] 2010. Dostupné z

<http://openvidia.sourceforge.net/index.php/Direct_Compute_Shader_Code_Listing>.

[15]**Khronos.** The OpenCL Specification. *The Khronos Group is a member-funded consortium focused on the creation of royalty-free open standards for parallel computing, graphics and dynamic media on a wide variety of platforms and devices.* [Online] 2009. Dostupné z <http://www.khronos.org/registry/cl/specs/opencl-1.0.48.pdf>.

[16]**Smith, Ryan.** NVIDIA's GeForce GTX 480 and GTX 470: 6 Months Late, Was It Worth the Wait? *AnandTech.* [Online] 2010. Dostupné z <http://www.anandtech.com/show/2977/nvidia-s-geforce-gtx-480-and-gtx-470-6-months-late-was-it-worth-the-wait-/6>.

[17]**IEEE.** IEEE 754: Standard for Binary Floating-Point Arithmetic. *IEEE Standards Working Group Areas.* [Online] 2008. Dostupné z <http://grouper.ieee.org/groups/754/>.

[18]**Microsoft.** COM Callable Wrapper. *MSDN: Microsoft Development, MSDN Subscriptions, Resources, and More.* [Online] Dostupné z <http://msdn.microsoft.com/en-us/library/f07c8z1c%28VS.71%29.aspx>.

[19]**Olmi, Eros a Bianchi, Roberto.** Basic Programming Language :: ThinBasic. [Online] 2010. Dostupné z <http://www.thinbasic.com/>.

[20]**Microsoft.** LoadLibrary Function (Windows). *Domovská stránka webu MSDN.* [Online] 2010. Dostupné z <http://msdn.microsoft.com/en-us/library/ms684175%28VS.85%29.aspx>.

[21] **Microsoft.** GetProcAddress function (Windows). *Domovská stránka webu MSDN.* [Online] 2010. Dostupné z <http://msdn.microsoft.com/en-us/library/ms683212%28VS.85%29.aspx>.

[22] **Microsoft.** FreeLibrary function (Windows). *Domovská stránka webu MSDN.* [Online] 2010. Dostupné z <http://msdn.microsoft.com/en-us/library/ms683152%28VS.85%29.aspx>.

[23]**Khronos.** Khronos OpenCL API Registry. *The Khronos Group: Open Standards, Royalty Free, Dynamic Media Technologies.* [Online] 2010. Dostupné z <http://www.khronos.org/registry/cl/>.

[24]**Žára, J., a další.** *Moderní počítačová grafika.* Brno : Computer Press, 2004. ISBN 80-251-0454-0.

[25]**Coufal, Jan.** *Detekce cesty pro mobilní robot analýzou obrazu.* Brno : Vysoké učení technické v Brně, Fakulta strojního inženýrství, 2010.

[26]**Schreiber P., Ondroušek V., Věchet S., Krejsa J.** *Parallelizing the Precomputed Scan Matching Method for Graphics Card processing.* Bratislava : RIE, 2010. in print.

[27]**Krejsa J., Věchet S., Hrbáček J., Schreiber P.** *High Level Software Architecture for Autonomous Mobile Robot. In Recent Advances in Mechatronics.* Berlin : Springer, 2009. stránky p. 185 - 190. ISBN 978-3-642-05021-3.

[28]**Advanced Micro Devices, Inc.** Software Developer Help Request Form . *Welcome to AMD Developer Central.* [Online] 2010. Dostupné z <http://developer.amd.com/support/KnowledgeBase/pages/HelpdeskTicketForm.aspx?Category=2&SubCategory=43>.

[29]**Nvidia.** Nvidia Online. *Nvidia Developer Web Site.* [Online] 2009. Dostupné z <https://nvdeveloper.nvidia.com/login.asp>.

[30]**Russel, Jordan.** Inno Setup. *jrsoftware.org // Jordan Russell's Software.* [Online] 2010. Dostupné z <http://www.jrsoftware.org/isinfo.php>.

SEZNAM PŘÍLOH

Příloha A – Zprovoznění realizovaného software
Příloha B – Přiložené CD-R médium s následujícím obsahem:

- Text této práce ve formátu PDF
  Dostupné z kořenového adresáře

- Video prezentace metody Precomputed Scan Matching
  Dostupné z adresáře /Video/

- Kód low level knihovny
  Dostupné z adresáře /Software/LowLevel/

- Šablona kódu knihovny pro zpracování obrazu a ukázková aplikace
  Dostupné z adresáře /Software/ImageProcessing/

- Aplikace demonstrující výpočet Precomputed Scan Matching na GPU
  Dostupné z adresáře /Software/PCSM/

- Generátor kódu pro OpenCL aplikace
  Dostupné z adresáře /Software/Generator/

PŘÍLOHA A: ZPROVOZNĚNÍ REALIZOVANÉHO SOFTWARE

Ke spuštění software na přiloženém médiu je vyžadováno PC s moderní grafickou kartou. Autor doporučuje počítač vybavený následujícími modely, které byly v průběhu realizace využity k testování:

- Nvidia GeForce 8800GT
- Nvidia GeForce 9500GT
- Nvidia GeForce 9800
- Nvidia GeForce GTX260
- Nvidia GeForce GTX275
- ATi Radeon HD 5770

V případě karet Nvidia postačuje instalace posledního ovladače, dostupného z: http://www.nvidia.com/Download/index5.aspx?lang=en-us.

V případě karet ATi je nutné instalovat kompletní ATi Stream SDK, dostupné z: http://developer.amd.com/gpu/ATIStreamSDK/Pages/default.aspx.

V instalovaných souborech je pak potřeba vyhledat knihovnu *atiocl.dll* a tu nakopírovat pod názvem *OpenCL.DLL* do adresáře s programem, který chcete spustit.

Programy není možné spouštět přímo z přiloženého média.